THÈSE

présentée par

Nina Tayar

pour obtenir le titre de

Docteur de l'Université Joseph Fourier
Grenoble I

(Arrétés ministériels du 5 juillet 1984 et du 30 mars 1992)

Spécialité

Informatique

Gestion des versions pour la construction incrémentale et partagée de bases de connaissances

Soutenue le 21 septembre 1995 devant le jury composé de :

Mme	Marie-Françoise	Bruandet	Président
MM.	Mourad	Oussalah	Rapporteur
	Christophe	Roche	Rapporteur
	Noureddine	Belkhatir	Examinateur
	François	Rechenmann	Directeur de thèse

Thèse préparée au sein du laboratoire LIFIA/IMAG

Remerciements

Toute ma reconnaissance et tous mes remerciements s'adressent à chaque personne ayant accepté de juger ce travail, m'ayant aidée au cours de la réalisation de cette thèse ou tout simplement m'ayant supportée durant ces années. Aussi je tiens à remercier :

Monsieur François Rechenmann, Directeur de recherche à l'INRIA, pour avoir dirigé mes recherches, pour ses conseils et ses encouragements. Qu'il trouve ici l'expression de ma gratitude.

Madame Marie-Françoise Bruandet, Professeur à l'Université Joseph Fourier, pour l'honneur qu'elle me fait de présider le jury.

Monsieur Mourad Oussalah, Professeur à l'EERIE de Nîmes et Monsieur Christophe Roche, Professeur à l'Université de Savoie, pour avoir accepté la lourde tâche d'être rapporteurs.

Monsieur Noureddine Belkhatir, Maître de Conférence à l'Université Pierre Mendès-France, pour l'attention qu'il a porté à mon travail et pour nos discussions toujours enrechissantes. Je lui sais gré de participer à mon jury.

Monsieur Philippe Jorrand, Directeur du LIFIA, de m'avoir accepté au sein de son laboratoire. Je tiens également à remercier l'ensemble du personnel administratif (Laurence, Danièle, Françoise, Lionel) pour leur sympathie et leur aide ponctuelle.

Cécile Capponi, ma collègue de bureau, qui a bien supporté mes sautes d'humeur, et qui a eu le mérite de m'apprendre à jouer au *Softball*.

Rubby Casallas-Gutierrez, qui a su répondre et fournir d'excellents pointeurs à mes questions sur les versions dans le domaine du Génie Logiciel.

Je remercie également tous les membres actuels de l'équipe Sherpa : Florence, Ysabelle, Sueli, Jutta, Danièle, Gilles, Jean-Yves, Jérôme, Jérôme, Pierre, Petko, Jean-Marc, Pierre, Patrice et les membres qui ont déjà quitté l'équipe : Nathalie, Olga, Alejandro, Alain, Mathias, Olivier et Bruno qui m'ont, à des titres divers et à des phases différentes, apporté le soutien nécessaire pendant ces années, et qui ont su me montrer que l'amitié peut dépasser les continents.

Je ne pourrais pas terminer cette liste d'hommages sans saluer toutes les personnes qui ont lu et relu ce mémoire et m'ont aidé à améliorer la forme et le fond de ce document : Gilles, Olivier, Patrice, Danièle, Rubby, Florence, Ysabelle, Monique, Lucien et Patrick.

Aux membres de la communauté syrienne à Grenoble, Maria, Rushed, Mazen, Hayate, Nawar, H.-Ali, Baidaa, S.-Ali, Mouna, Imad, Riad, Kinda, Arije, Oussama, Mouna, Adnan, Ramez, Joumana, en espérant n'avoir oublié personne, que je remercie du fond du cœur pour leur aide inestimable à relativiser la distance entre la France et mon pays: "shoukran lakom" .

Je n'oublie pas de remercier très chaleureusement mes amis français, que je regrette de ne pouvoir tous citer, et dont la rencontre a contribué à apprecier encore plus la splendeur de la région Rhône-Alpes : ski, randonnées, soirées, restos, cinémas, théâtre...

Enfin et surtout, ma famille et mes proches pour m'avoir permis d'arriver jusqu'à ici et pour leur confiance quant à ma réussite. Qu'ils trouvent ici le témoignage de ma plus sincère gratitude.

Nina

Résumé

Dans de nombreux domaines scientifiques ou techniques, la quantité et la complexité croissantes des connaissances manipulées rendent inadéquate l'utilisation de supports classiques tels que le papier. L'informatique peut apporter une réponse en proposant un cadre pour modéliser, gérer et exploiter ces connaissances : on parle alors de *bases de connaissances*.

Pour la plupart des domaines, la connaissance n'est pas statique et immédiatement disponible mais est au contraire en constante évolution au fur et à mesure des découvertes. Dès lors, une base de connaissances est construite de manière incrémentale. Incrémentale veut dire que l'on commence par construire une petite base qui est enrichie petit à petit par l'acquisition de nouvelles connaissances. Cette construction est partagée puisque plusieurs personnes doivent pouvoir travailler simultanément sur cette base afin de la bâtir.

Cette thèse aborde la problématique de la gestion des versions pour les bases de connaissances. Ses apports sont de deux ordres. Nous avons tout d'abord conçu et réalisé un système de gestion de versions de bases de connaissances. Par définition, une version reflète un état de l'évolution de la base, c'est-à-dire de l'ensemble des structures et des valeurs des connaissances contenues dans celle-ci (à un moment donné). Les versions aident ainsi à contrôler l'historique des changements effectués au cours de la construction de la base. Elles permettent également un éventuel retour en arrière. Finalement, les versions permettent la formulation de différentes hypothèses de recherche et la gestion de leur évolution parallèle au cours du temps.

Nous avons également contribué à la conception de l'environnement de construction incrémentale et partagée de bases de connaissances et montré comment notre système de versions s'intègre au sein de cet environnement.

Mots-clefs

Base de connaissances, construction incrémentale et partagée de bases de connaissances, connaissances consensuelles, représentation de connaissances à objet, version d'instance, version de classe, version de base de connaissances.

Abstract

For major scientific and technical domains, traditional media (e.g. paper) are insufficient to handle the increasing volume and complexity of knowledge. Computers may offer efficient alternatives for the modelling, management and exploitation of knowledge. Such solutions are known as *knowledge bases*.

In most domains, knowledge is not static and complete but rather evolving and being discovered. Consequently, a knowledge base must be built incrementally over time. Incremental means that we start with a small base, which grows with the acquisition of new knowledge. Many people may need to work together to build the knowledge base.

This thesis studies the version management problem in the context of knowledge bases. It offers two main contributions. Firstly, we design and implement a version management system for knowledge bases. By definition, a version is a state of the base at a precise moment. Versions help to maintain a history of changes. They also allow decisions to be reversed and alternative hypotheses to be investigated.

Secondly, we contribute to the design of an environment for incremental and concurrent knowledge base building. We point out how our version system can be integrated into this environment.

Keywords

Knowledge base, incremental and concurrent knowledge base building, consensual knowledge, object-based knowledge representation language, instance version, class version, knowledge base version, computer-aided collaborative research.

Table des matières

Chapitre 1

Introduction

Dans de nombreux domaines scientifiques ou techniques, la quantité et la complexité croissantes des connaissances manipulées rendent inadéquate l'utilisation de supports classiques tels que le papier. Ce problème est rencontré par exemple dans le cadre des recherches menées sur le décodage du génome humain faisant apparaître la nécessité de concevoir et de développer des outils appropriés [Schmeltzer et al., 1993]. L'informatique peut apporter une réponse en proposant un cadre pour modéliser, gérer et exploiter ces connaissances : on parle alors de *systèmes à bases de connaissances* (SBC).

1.1 Bases de connaissances

Un domaine majeur de l'Intelligence Artificielle (IA) est la conception des systèmes déclaratifs, communément appelés systèmes à bases de connaissances. Contrairement à la programmation procédurale, ces systèmes sont caractérisés par une séparation entre les connaissances nécessaires pour résoudre un problème et les mécanismes exploitant ces connaissances. Cette séparation permet de décrire les connaissances indépendamment de leur utilisation ultérieure. Ceci facilite d'une part la modification et l'ajout de nouvelles connaissances à la base. D'autre part, on peut fournir des justifications et des explications du comportement du système [Rechenmann, 1993]. Un système à base de connaissances est donc constitué de deux éléments : une base de connaissances (BC) où les connaissances relatives à un domaine sont réunies et des mécanismes de raisonnement (moteurs d'inférence) qui iront puiser les connaissances qui leur sont nécessaires afin de résoudre un problème particulier ou de mener au but recherché.

1.2 Construction d'une base de connaissances

Les connaissances présentes au sein d'une BC sont fournies la plupart du temps par un expert du domaine. Lorsque ce domaine est parfaitement maîtrisé, l'expert possède l'ensemble des connaissances qu'il va devoir transmettre au système. Sa tâche consiste alors à les sélectionner et à les modéliser afin de les rendre utilisables par la machine.

Dans le cadre de la recherche scientifique, domaine auquel nous nous sommes intéressés au cours de notre travail, la connaissance n'est pas statique et immédiatement disponible mais est au contraire en constante évolution au fur et à mesure des découvertes. Deux remarques importantes peuvent être faites concernant le processus de création de cette connaissance :

1. La progression scientifique et donc l'augmentation des connaissances résulte de l'interaction entre les nombreux chercheurs impliqués dans une même communauté scientifique (et plus particulièrement dans un même projet). Cette interaction a pour objectif d'échanger les différentes idées et d'essayer de parvenir à un accord (ou consensus) sur les résultats expérimentaux et leurs interprétations.

2. Au fur et à mesure que de nouvelles expériences sont réalisées, les chercheurs améliorent leurs connaissances du domaine. De nouvelles hypothèses peuvent alors être formulées, d'anciennes peuvent être abandonnées ou ré-étudiées selon un nouvel axe. Aussi, l'activité de recherche nécessite de faire des retours en arrière.

Les deux aspects précédemment cités sont au cœur de la recherche scientifique et doivent donc être pris en compte au sein de tout outil informatique destiné à aider à partager les connaissances attachées à un domaine de recherche. Partager des connaissances communes revient à partager le processus de construction d'une base de connaissances consensuelles [Rechenmann, 1994].

1.3 Construction d'une base de connaissances consensuelles

Dans ce travail, nous nous plaçons dans le cadre du système de construction d'une base de connaissances consensuelles proposé par Rechenmann [Rechenmann, 1993] et repris par Eusenat [Euzenat, 1995]. Ce système repose sur le cycle soumission-validation-correction rencontré lors de la publication d'un article (voir fig. 1.1).

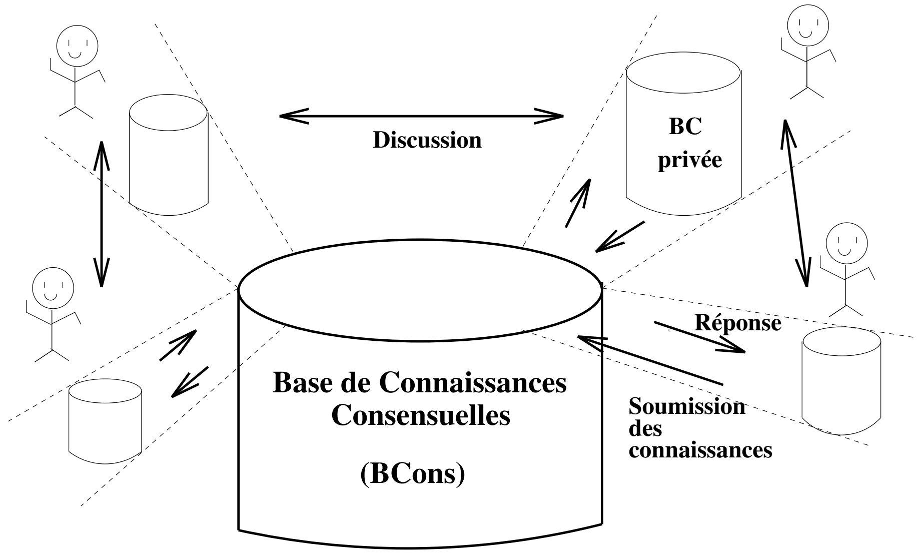


Figure 1.1 - : Le processus de construction d'une base consensuelle repose sur la métaphore de la soumission d'un papier : soumission-validation-correction.

Le principe est de construire une base centrale (appelée BCons) contenant un ensemble de connaissances consensuelles provenant des bases privées de chacun des chercheurs impliqués dans ce processus. Lorsqu'un chercheur dispose d'une "théorie" personnelle, il n'est assuré que très localement de sa pertinence. Afin de la valider, il va la soumettre à la BCons afin de confronter cette nouvelle connaissance à l'ensemble des connaissances déjà présentes. La fin de cette étape est signalée par une réponse sous forme d'un rapport contenant le résultat de cette évaluation. La réponse peut être positive et les connaissances seront incorporées dans la BCons et par conséquent partagées avec les autres chercheurs. La réponse peut également être négative. Des corrections sont alors demandées. Le chercheur concerné peut accepter le rapport et corriger ses connaissances dans l'intention de les soumettre à nouveau. Il peut également refuser le rapport et demander d'établir une discussion avec ses collègues (les autres constructeurs de la BCons) afin de confronter ses connaissances avec les leurs. Ces multiples confrontations vont favoriser l'émergence d'un accord sur les connaissances intégrées dans la BCons.

1.4 Problème abordé

Ce système, tel que nous venons brièvement de le décrire, permet la construction d'une base de connaissances consensuelles par la mise en commun d'un ensemble de connaissances individuelles que possède ou acquiert un ensemble de chercheurs impliqués dans un projet commun. Il répond à ce titre à la première propriété énoncée au cours de la section 1.2. Il ne prend par contre pas en compte le second aspect énoncé et ne permet donc pas de gérer un des aspects très importants du mode de travail d'un chercheur : la formulation d'hypothèses de recherche et la gestion de leurs évolutions au cours du temps. Aussi, nous nous sommes intéressés à la conception et à la réalisation au sein du système existant d'une approche permettant de prendre en compte cet aspect de la recherche. Nous pouvons dégager trois types de besoins :

1. Gestion de l'évolution des connaissances de chercheurs au cours du temps :

   Nous avons déjà vu que cette évolution est une caractéristique intrinsèque de la recherche scientifique. Nous avons besoin de la gérer (i.e. garder ses traces) afin que les différents chercheurs impliqués dans la construction de la BCons puissent par exemple comprendre l'avancement de la science dans un domaine donné ou puissent également comprendre les changements qu'une entité particulière a subis au cours du temps. Le système pour cela doit être capable de répondre à des interrogations du type : *Comment les recherches ont permis d'arriver à un contenu actuel de la BCons? Pourquoi un tel contenu? Quelle était la composition précédente de l'entité génome?* etc.

2. Gestion de la construction incrémentale de la BCons :

   Une base de connaissances est rarement élaborée en une seule étape. Son processus de construction est une succession d'opérations de modifications, d'additions et de suppressions de connaissances. Ces opérations vont permettre d'améliorer la base et de la rendre la plus complète possible. Nous avons besoin de gérer cet aspect d'incrémentalité en offrant la possibilité à la communauté de chercheurs de réviser leurs “anciennes” connaissances, de reconsidérer des hypothèses abandonnées ou d'expérimenter de nouvelles hypothèses tout en gardant les traces de chaque changement effectué (voir le point précédent).

3. Gestion de la construction partagée de la BCons :

Comme nous l'avons signalé auparavant, plusieurs chercheurs interviennent dans le processus de construction. Ils travaillent en parallèle afin de progresser dans leur propre recherche. Ils essayent ensuite de partager le fruit de leur avancement. Les résultats finaux de leur recherche peuvent être différents sans être cependant contradictoires (par exemple, différentes modélisations possibles d'une entité étudiée). Le système a alors besoin d'inclure au sein de la BCons tous ces résultats à condition qu'un accord soit donné par tous les chercheurs concernés. Ainsi, il n'y a pas de conflit entre la gestion des différents résultats des chercheurs (concernant des connaissances particulières du domaine étudié) et l'aspect consensuel de la base élaborée.

Parallèlement à ces problèmes, l'utilisateur de notre système n'étant pas forcement informaticien, il faut que les différents mécanismes de gestion de l'évolution de connaissances soient les plus naturels et les plus transparents possibles.

1.5 Apports de ce travail

Ce travail a pour objectif de répondre aux besoins exposés au cours de la section précédente. Notre contribution est de deux ordres. Nous avons tout d'abord conçu et réalisé un système de gestion de versions de bases de connaissances. Par définition, une version d'une base de connaissances reflète un état de l'évolution de cette base (à un moment donné). Nous définissons un état comme l'ensemble des structures et des valeurs des connaissances de la base. Le modèle choisi pour représenter les connaissances est le modèle à objets. Notre système permet de répondre aux besoins précédemment cités grâce aux caractéristiques suivantes :

1. La gestion de l'historique des changements effectués au cours de la construction de la BCons :

   Une version est créée chaque fois qu'une opération de modification est exécutée. L'ensemble des versions de la base constitue son historique.

2. La gestion des arbres de versions :

   Les différents résultats expérimentaux ou les différentes hypothèses de travail sont supportés par l'intermédiaire des branches disjointes de l'arbre de version.

3. La gestion de l'évolution des connaissances au niveau des valeurs et des structures de données :

   Dans un modèle à objets, nous parlons de la coexistence de plusieurs versions d'instances et de classes. La gestion des différentes versions d'une instance permet de garder l'historique de son évolution. Dans le cas des classes, elle permet en outre la cohabitation des anciennes classes (et leurs anciennes instances) avec les nouvelles classes (et leurs nouvelles instances). Ceci signifie que le changement d'une classe n'a pas d'impact sur ses anciennes instances qui peuvent toujours exister et être utilisées par les intéressés. Notre système propose (contrairement aux autres travaux de la littérature) une gestion uniforme dans les deux cas d'évolution. Cela signifie, en particulier, que les modifications de classes sont gérées de la même façon que celles effectuées sur les instances. Toute modification génère une nouvelle version de l'entité constituée de la structure et de la valeur de l'entité. Ce point représente une des originalités de notre système.

4. La transparence du système de versions :

   Les relations entre les versions et les bases sont gérées automatiquement. Une fois la version déterminée, l'utilisateur ne perçoit du système que la base correspondante. Il peut donc continuer à travailler comme il le faisait auparavant sans les versions.

5. La prise en compte des aspects particuliers des bases de connaissances dans la gestion de leurs versions :

   Comme nous l'avons vu dans 1.1, les SBC sont constitués de deux éléments : les connaissances et les mécanismes d'inférence pouvant produire de nouvelles connaissances. Des mesures particulières ont été prises afin d'intégrer l'impact de ces mécanismes sur la gestion de la cohérence des versions de la base. Cette caractéristique de notre système est également originale, les approches existantes s'intéressant aux versions dans les bases de données et non pas dans les bases de connaissances.

Nous avons également contribué à la conception de l'environnement de construction de bases consensuelles ainsi qu'à l'intégration de notre système de versions au sein de cet environnement. Cette intégration nous a permis d'obtenir un système global permettant d'assister un chercheur dans la gestion de ses connaissances tout en respectant son mode habituel de travail.

1.6 Plan du mémoire

Ce mémoire comporte les quatre chapitres principaux suivants :

Chapitre 2

Nous réalisons un tour d'horizon sur la problématique de la gestion des versions. Nous étudions les systèmes dont les caractéristiques sont proches des nôtres (évolution dans le temps, travail en groupe, etc.). Nous nous intéressons ainsi aux bases de données temporelles ainsi qu'aux systèmes ayant une activité de conception (bases de données dédiées à la CAO et au génie logiciel).

Chapitre 3

Notre système de gestion de versions est détaillé dans ce chapitre. Nous décrivons l'architecture proposée pour modéliser les versions de la base. Nous détaillons ensuite l'ensemble des opérations fournies permettant de les gérer. Nous abordons également la manipulation des connaissances de la base à travers les versions. Nous achevons le chapitre par une description détaillée du contrôle de la cohérence de chaque version.

Chapitre 4

Ce chapitre explique comment notre système, développé dans le chapitre précédent, va aider à la construction incrémentale et partagée de la BCons. Nous commençons par représenter l'environnement développé pour l'élaboration de la BCons en décrivant ses composants. Nous distinguons entre trois types de bases : consensuelle, de collaboration et de travail. Ces trois types représentent des étapes de transformation et d'affinement des connaissances des chercheurs. Ils facilitent, entre autres, l'obtention du consensus.

Finalement nous abordons l'intégration du module de gestion des versions au sein de l'environnement en détaillant ses multiples tâches.

Chapitre 5

Ce chapitre est consacré à la réalisation de notre gestionnaire de versions, aboutissant au prototype *VOG*. Toutes les structures utilisées pour modéliser les versions y sont présentées. La mise en œuvre des opérations de manipulation des versions est également abordée. La dernière section de ce chapitre décrit le langage de requêtes proposé. Ce langage contient des opérateurs temporels permettant d'interroger l'ensemble des versions d'une base.

Chapitre 2

Les versions et leurs systèmes de gestion

De nombreuses études dans des domaines diverses de l'informatique se sont intéressées aux problèmes de la gestion de plusieurs versions d'une entité. Une version est par définition un état[1] d'une entité que le système ou l'application veut conserver. Deux niveaux d'utilisation des versions existent :

- niveau-système : les versions sont directement créées et utilisées par le système de bases de données. Elles sont considérées comme un moyen pour contrôler ainsi que synchroniser l'accès aux entités partagées [Agrawal et Sengupta, 1989], résister aux pannes [Bernstein et al., 1987] ou encore améliorer les performances dans les systèmes distribués [Weihl, 1987]. Dans la plupart de ces cas, les versions (dans ce cas on peut plutôt les appeler "copies") sont utilisées pour une durée limitée, le temps nécessaire pour valider les changements effectués sur la base. Elles sont cachées à l'utilisateur qui continue à travailler dans un univers mono-version.

- niveau-application : les versions sont créées à la demande des utilisateurs pour des fins spécifiques.

Cette thèse est consacrée exclusivement au niveau-application que nous détaillons par la suite.

L'objectif de notre travail est le développement d'un système de gestion de versions des bases de connaissances. Ce sujet étant très peu abordé dans la littérature consacrée aux bases de connaissances, nous nous sommes tournés vers les travaux développés dans le cadre des bases de données où ce problème a été particulièrement traité.

Les besoins détaillés dans le chapitre précédent (Cf. § 1.4) nous ont permis de limiter cet état de l'art aux bases de données d'aide à la conception et aux bases de données temporelles[2]. Les caractéristiques de ces deux catégories de systèmes nous semblent les plus proches des nôtres. Nous détaillons ces caractéristiques dans la section suivante.

[1] La valeur ou la description.

[2] Nous ne faisons pas la différence entre les deux termes bases temporelles et bases historiques (voir [Tansel et al., 1993] pour plus de détails sur les deux termes). Ici, on considère que les deux sont des synonymes et désignent des bases de données où la dimension du temps a été introduite

2.1 Principales caractéristiques des systèmes étudiés

Les systèmes étudiés sont ceux de génie logiciel, de bases de données dédiée à la Conception Assistée par Ordinateur (CAO) et de bases de données temporelles. Ces systèmes ont été choisis car ils ont les caractéristiques suivantes :

1. le processus de conception est basé sur un principe "essai-erreur" [Fauvet, 1988], [Trousse, 1989] et [Gançarski, 1994]. Le concepteur crée une version initiale du produit. Il essaye ensuite de l'améliorer afin de converger vers sa spécification finale et complète. Il peut cependant avoir besoin de revenir à une conception antérieure s'il constate qu'il devait procéder autrement ou s'il souhaite essayer une autre méthode et comparer les produits finaux (cette caractéristique rejoint la caractéristique d'incrémentalité dans notre cas).

2. la conception met en jeu des entités de taille importante (par exemple les logiciels dans les ateliers de génie logiciel) nécessitant l'intervention de plusieurs personnes. Dès lors, des données sont partagées et d'autres sont échangées ou manipulées simultanément par les équipes de développement [Belkhatir, 1988] (cette caractéristique rejoint celle de partage de processus de construction).

3. les entités conçues (circuits, logiciels, bâtiments ou même images) sont composées d'autres entités qui sont elles-mêmes des entités à concevoir [Talens et al., 1993]. Par conséquent, le modèle de données dans les systèmes de conception est un modèle hiérarchique. Il manipule des structures de nature complexe (dans notre cas, les connaissances sont aussi de nature complexe).

4. les structures des entités conçues peuvent être sujet à des changements pendant leur durée de vie. Cette évolution est une conséquence de plusieurs facteurs [Ahmed-Nacer, 1994] : (1) les besoins exprimés par une application sont rarement stables, (2) les besoins des utilisateurs d'expérimenter plusieurs structures de données avant de choisir celles qui leur conviennent, (3) les besoins d'étendre le domaine d'application de la base de données [Andany et al., 1991]. Tous ces facteurs nécessitent la modification des structures de données afin de les adapter à de nouveaux traitements (dans notre cas, nous avons également besoin de l'évolution des descriptions des connaissances).

5. les entités manipulées par certaines applications évoluent temporellement. Cette caractéristique a mis en évidence le besoin de gérer le temps [Quang, 1986]. Pour ces applications, connaître l'état de la base de données par rapport à une date précise est important. Dans le domaine de la bureautique, par exemple, l'administrateur de la base de données peut être intéressé par les anciennes valeurs ou par l'historique de quelques informations (le salaire ou la situation familiale d'un employé).

La gestion des versions multiples des entités a rapidement été perçue comme nécessaire dans de tels systèmes. En génie logiciel, [Rochkind, 1975], [Tichy, 1982], [Kaiser et Habermann, 1983] et [Estublier, 1985] sont un échantillon des premiers travaux produits. Pour les bases de données dédiées à la CAO, l'utilisation des versions a commencé à être une direction importante de recherche depuis les travaux de Katz et Lehmann [Katz et Lehmann, 1984] et ceux de Kim et Batory [Kim et Batory, 1985]. Les premières publications sur l'incorporation de la notion du temps dans les bases de données sont celles de

[Wiederhold et al., 1975] et [Bubenko, 1977].

Dans la suite, nous aborderons les différentes approches proposées afin de répondre aux attentes de ces trois communautés. Nous allons pour cela nous intéresser aux techniques de manipulation des versions d'entité existantes dans la littérature en nous concentrant en particulier sur les cas où les entités sont représentées selon le formalisme à objets (c'est le formalisme choisi pour représenter les connaissances dans notre cas et qui sera présenté dans le chapitre 3).

Ce chapitre est organisé de la façon suivante : la section 2.2 décrit la problématique de versions relative au modèle de données à objets. Les principaux termes liés aux versions sont définis au cours de la section 2.3, permettant de préciser la terminologie utilisée tout au long de ce chapitre. Les différents aspects de la problématique de versions sont étudiés à partir de la section 2.4. Les aspects qui nous intéressent sont ceux liés à la modélisation des versions et à leur gestion. Les versions peuvent être soit des versions d'objets (simples ou complexes) soit des versions de types. C'est pourquoi, nous avons développé trois sections différentes (2.5, 2.6, 2.7), traitant respectivement la modélisation et la gestion de ces trois groupes de versions. La section 2.8 conclut ce chapitre et signale les limites des solutions proposées.

2.2 Versions et modèle à objets

Notre travail se situe dans le cadre des modèles de données à objets. Nous étudions la problématique de *versionnement*[3] relative à ce modèle. Nous introduisons brièvement, dans cette section, les notions principales rencontrées dans un modèle à objets traditionnel.

Un objet représente une entité du monde réel. Chaque objet est une instance d'un type. Un type[4] est une structure de données qui définit un ensemble de propriétés (appelées attributs) pour ses instances, et un ensemble d'opérations (appelé interface) appliquées à ses instances. Les types sont organisés dans une hiérarchie. Un type T hérite les propriétés et les opérations additionnelles de son type supérieur. Un attribut dans un type $T1$ peut avoir comme valeur un autre type $T2$, définissant de cette manière une relation de référence entre les objets de $T1$ et ceux de $T2$. On dit ainsi qu'un objet de type $T1$ est complexe et contient des liens vers d'autres objets (appelés objets référencés) de type $T2$. Dans la plupart des systèmes de conception, ce lien indique souvent une relation de composition. Nous utilisons alors le terme d'objet composite (i.e. composé d'objets composants). Chaque objet dans la base possède un identificateur unique permettant de le référencer dans d'autres objets.

Dans ce cadre, l'aspect qui nous intéresse est l'évolution des objets et des types. Chaque type peut changer dans sa définition (l'ensemble de ses attributs) ou dans son interface. Aussi, les valeurs des attributs dans chaque objet peuvent être modifiées, supprimées ou ajoutées. L'objectif souhaité est de contrôler ces changements en associant des versions aux objets et aux types de la base. La problématique est expliquée schématiquement à la figure 2.1. Une telle situation exige la résolution de certains problèmes (expliqués figure 2.1) dus à l'addition de la notion de version au modèle de données. Le système de gestion de base de données doit permettre à ses utilisateurs de représenter, d'accéder et de manipuler les entités multi-versions.

[3] Versioning en anglais.

[4] Le terme classe sera, dans notre cas, utilisé comme synonyme du terme type.

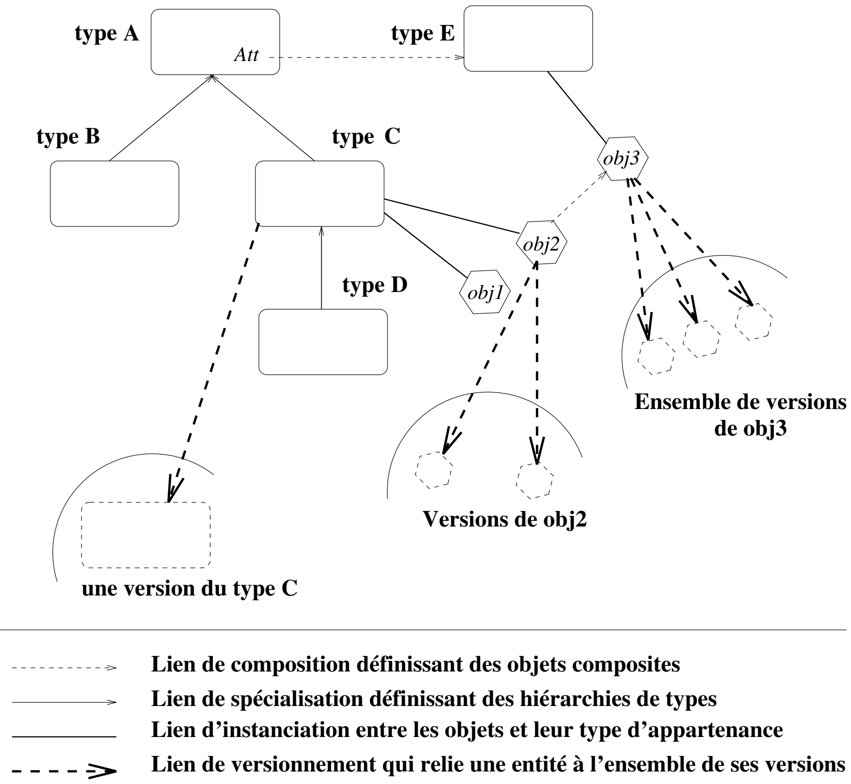


Figure 2.1 - : La problématique de versionnement dans un modèle de données à objets. Un objet appartient à un type et peut être composé d'autres objets. Il peut avoir plusieurs versions ainsi que son type et ses objets composants. Une telle situation nécessite de nouveaux outils afin d'une part de représenter les relations entre les objets et leurs versions, les relations entre les versions d'un objet et les versions de ses objets composants et les relations entre un objet et les versions de son type. D'autre part, ces outils doivent permettre d'accéder et de manipuler ces entités multiples.

Dans ce qui suit, nous étudions les différentes approches pour la gestion des versions dans un formalisme orienté-objet. Malgré tous les travaux effectués sur ce sujet, la terminologie utilisée est différente suivant les domaines. C'est pourquoi, nous allons tout d'abord préciser la définition des termes principaux utilisés dans le versionnement d'une entité (un objet ou un type).

2.3 Définition des termes principaux du versionnement d'une entité

Historique d'une entité: c'est l'évolution de cette entité au cours du temps. Il s'agit donc de l'ensemble de ses versions successives ordonnées en fonction du temps.

Hiérarchie de versions: elle définit la relation entre les différentes versions d'une entité. Une version $V2$ est appelée une *version-dérivée* (ou *révision*) de $V1$ si elle succède directement à cette dernière dans la même branche. Par contre, $V3$ et $V4$ sont deux *variantes* si elles appartiennent à deux branches différentes et sont dérivées de la même version[5] (voir fig. 2.2). Pour les applications de conception, chaque

[5] Elles sont également dites *concurrentes* car elles existent simultanément.

branche dans l'arbre de versions représente un choix de conception (nouvel outil, nouvelle technologie, etc.). Cette relation de prédécesseur-successeur permet, entre autres choses, de retrouver une version à partir de ses ascendantes.

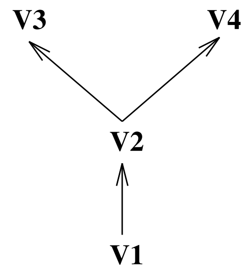


Figure 2.2 - : Les révisions et les variantes dans une hiérarchie de versions. *V2* est une révision de *V1*. Les deux versions *V3* et *V4* sont deux variantes.

Cette hiérarchie peut se présenter sous forme d'une liste linéaire où les variantes n'existent pas (cas de la plupart des bases temporelles), sous forme d'un arbre où une version est dérivée d'une seule version mère (cas de la plupart des bases dédiées à la conception), et enfin sous forme d'un graphe lorsque le système de gestion offre la possibilité de fusionner deux versions en une seule (cas du système `Avance` [Björnerstedt et Hultén, 1989]).

Granularité de versionnement : c'est l'élément sur lequel s'applique le changement [Ahmed-Nacer, 1994]. La granularité n'est pas la même partout. Elle peut varier d'un objet élémentaire [Kim et al., 1987] jusqu'à une base de données [Cellary et Jomier, 1990]. Dans tous les cas, elle représente l'unité de cohérence. Lorsque la granularité est un objet, par exemple, chaque nouvelle version de l'objet doit être cohérente avec son type d'appartenance (un objet est dit cohérent s'il satisfait la description de son type).

Configuration d'un objet [6] : c'est l'association d'une version d'un objet composite (appelée version composite) avec les versions de ses composants [Tichy, 1988]. Une version composite peut être décrite par des configurations multiples chaque fois qu'elle référence une version différente d'un de ses composants : voir fig. 2.3 (dans le cas d'un objet qui est composé de n objets dont chacun peut avoir m versions, le nombre de configurations possibles est de m^n). Cependant, toutes les configurations possibles ne sont pas admissibles ou ne satisfont pas l'ensemble des contraintes de conception. C'est pourquoi, les configurations acceptées sont celles qui contiennent un ensemble de versions d'objets mutuellement consistantes (c'est-à-dire non contradictoires entre elles). De ce fait, une configuration pour quelques systèmes (par exemple [Katz, 1990]) est définie comme une unité de cohérence (la granularité de versionnement est alors supportée au niveau des configurations).

Suite à cette définition de termes, nous allons maintenant aborder les différents aspects de la problématique des versions. Nous commençons par celle d'objets, suivie par celle de types. Nous allons tout d'abord aborder la sémantique et la création de versions d'objets.

[6] La configuration d'un type complexe est similaire à celle d'un objet. L'étude est restreinte à celui-ci, sachant que les mécanismes exposés pour reconstituer des configurations d'objets peuvent également être appliqués à des types.

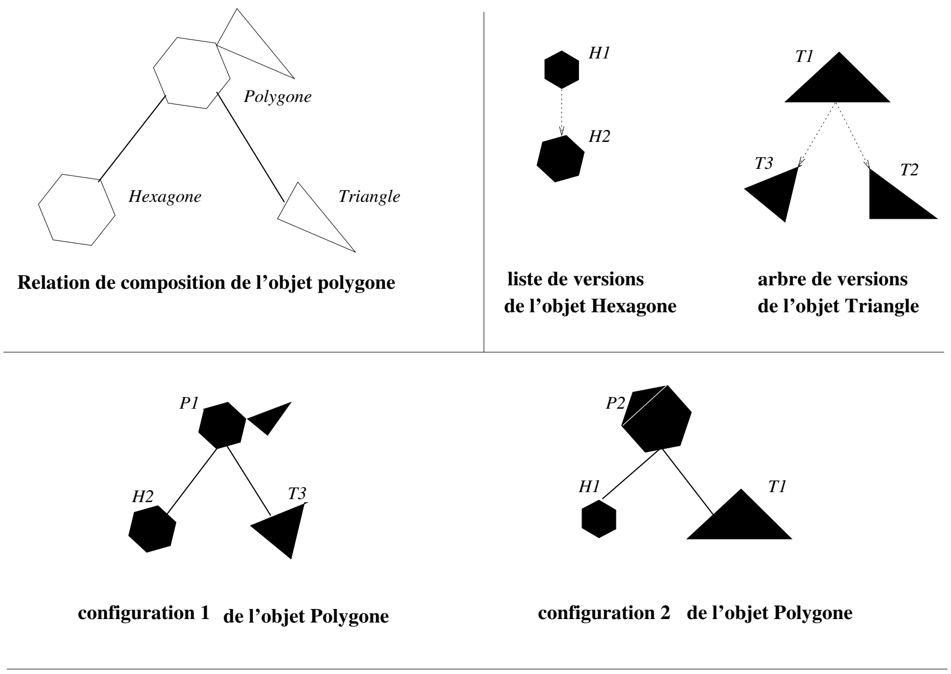


Figure 2.3 - : Configuration d'un objet composé de deux autres objets. Le premier ayant deux versions et le second possédant trois versions. Dans ce cas, l'objet composite peut avoir six configurations possibles.

2.4 Sémantique et création de versions d'objets

2.4.1 Sémantique

Il existe différentes sémantiques possibles pour la génération d'une nouvelle version. La sémantique la plus répandue et la plus précise est la temporelle : chaque opération de mise à jour de l'objet crée une nouvelle version de celui-ci. Cependant il existe des travaux (notamment dans le domaine du génie logiciel) qui insistent sur le fait que les versions ne sont pas simplement des données qui varient avec le temps, elles ont un sens beaucoup plus riche qui peut être différent d'une application à une autre. C'est pourquoi [Katz, 1990] et [Agrawal et al., 1991] conseillent de laisser le choix à l'utilisateur en lui fournissant des opérations de création explicite des versions de leurs objets.

Pour les applications ayant une sémantique a priori (par exemple, les bases temporelles où la sémantique est le temps [Tansel et al., 1993]), une entité qui varie selon celle-ci va avoir automatiquement une nouvelle version créée. En revanche, pour d'autres applications, un programmeur a la possibilité de réviser n'importe quelle version précédente de son logiciel afin de l'améliorer ou de l'adapter à des besoins additionnels. Pour ce faire, il va insérer les changements dans une nouvelle version dérivée de l'ancienne. De même, un programmeur peut développer deux variantes de son logiciel, une est codée en *Lisp* et l'autre en *Pascal*. Ces deux versions peuvent exister simultanément.

2.4.2 Création de versions

Lorsque la décision de création d'une nouvelle version est automatique (effectuée par le système), il est indispensable de définir les critères qui dirigent cette opération. Ceux-ci peuvent être exprimés en terme de temps en indiquant la *fréquence* de la création d'une version [Fauvet, 1989]. Par exemple, chaque jour ou chaque semaine, une nouvelle version d'un objet concerné doit être créée. Selon la fréquence choisie, cette obligation peut multiplier ou réduire le nombre de versions d'un objet. Dans le premier cas, elle risque de conduire à des versions identiques et, dans le second cas elle risque de ne pas préserver des changements majeurs de l'objet. D'autre part, les critères peuvent être définis par un ensemble d'attributs dont la mise à jour implique la dérivation d'une nouvelle révision de leurs objets. Cet ensemble est appelé la *sensibilité* d'une version (ou *attributs significatifs* [Ahmed et Navathe, 1991], *attributs sensitifs* [Talens, 1994]). Parmi ces attributs, il y a ceux appelés *attributs alternatifs* dont la modification crée une nouvelle variante de l'objet (ainsi une nouvelle branche dans la hiérarchie de versions), ce qui signifie que toutes les versions d'un objet ayant les mêmes valeurs de ces attributs appartiennent par définition à la même branche [Sciore, 1994]. Pour la conception d'une voiture, l'attribut *moteur* peut être défini comme un attribut alternatif. Une voiture peut avoir des versions avec un moteur diesel, et d'autres versions (variantes des premières) avec un moteur à essence.

Dans la plupart des modèles de génie logiciel (voir par exemple `Damokles` [Dittrich, 1989] et `Adèle` [Belkhatir et al., 1991]), les notions d'attributs sensibles et d'attributs alternatifs sont remplacées par les notions de modifications *mineures* (par exemple, la correction des erreurs de programmation[7], ou l'addition de nouvelles fonctionnalités) et modifications *majeures* (par exemple, le choix d'un nouveau système d'exploitation ou d'un nouveau langage de programmation). Ces modifications permettent alors de gérer l'évolution d'une version en révision ou en variante.

Une fois que les règles de génération de nouvelles versions sont bien spécifiées, il faut pouvoir, au niveau du système, distinguer entre la création des instances d'un type et la création des versions de ces instances. Il faut signaler que si un objet de type T a trois versions, ceci ne correspond pas à quatre instances de T. Pour concrétiser cette séparation, un nouveau niveau d'abstraction est ajouté, géré par la notion de type et d'objet génériques.

2.5 Modélisation et gestion des versions d'objets élémentaires

2.5.1 Objet générique

Un objet contient parmi ses attributs ceux qui permettent de l'identifier (appelés *invariants* [Ahmed et Navathe, 1991] ou *abstraits* [Kafer et Achöning, 1992]) et ceux qui peuvent varier d'une version à une autre de cet objet (les attributs *sensibles*). Le système de gestion définit un *type générique* comme un type dont les instances, dites *objets génériques*, sont immuables. Ces dernières contiennent les attributs communs à toutes leurs versions. Les versions d'un objet générique, qui sont elles-mêmes des instances d'un autre type, partagent la même définition (attributs et opérations) mais les valeurs de leurs attributs sont différentes [Dittrich et al., 1986], [Sciore, 1994]. Les autres systèmes parlent de notions similaires : *groupe de versions* [Tichy, 1988] et *famille de versions* [Wiebe, 1993] qui dénotent une progression du développement des états d'un *objet partiel* (l'équivalent de l'objet générique) ou bien un ensemble de variantes de celui-ci.

Chaque version contient en outre des attributs générés par le système permettant à la fois de stocker les informations propres à la version (numéro ou date de création) et

[7] Bugs.

d'exprimer sa relation avec les autres versions de son objet. Prenons par exemple les bases historiques, avec un type *Personne* ayant les attributs : *nom*, *no-sécurité-sociale*, *salaire* et *lieu-travail*. Le type générique va donc contenir les attributs non temporels (qui ne varient pas avec le temps) : *nom* et *no-sécurité-sociale*, tandis que le type version contiendra les propriétés temporelles : *salaire* et *lieu-travail* (voir fig. 2.4). Généralement, le type définissant les versions d'un objet est dénoté par <nom du type de l'objet>VERSION où <nom du type de l'objet> est le nom du type de son objet générique.

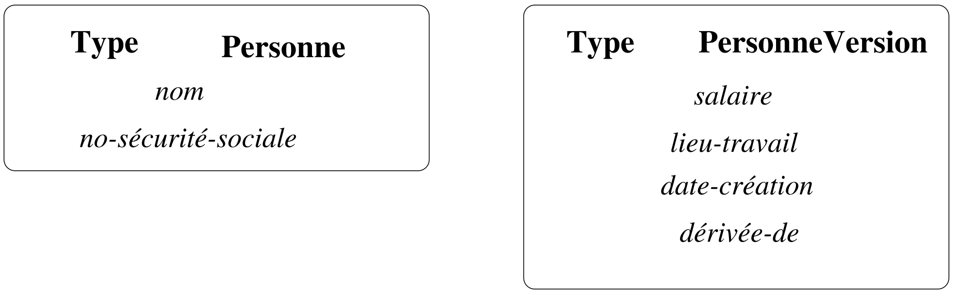


Figure 2.4 - : Le type générique *Personne* contient les attributs qui identifient une personne. Le type *PersonneVersion* contient les attributs qui prennent des valeurs différentes d'une version à une autre. Il contient aussi deux attributs *date-création* et *dérivée-de* permettant de gérer les versions.

2.5.2 Relation entre l'objet générique et ses versions

L'objet générique est le représentant de l'ensemble de ses versions [Chou et Kim, 1988]. De même, chaque version appartient exactement à un objet, son objet générique. Cette relation mutuelle est exprimée par deux attributs. Le premier, inséré dans le *type générique*, est une référence à l'ensemble des versions de l'objet concerné. Le deuxième, défini dans le *type version*, pointe sur le *type générique* associé. De ce fait, la figure 2.4 est remplacée par la figure 2.5.

Agrawal dans son système `Ode` [Agrawal et al., 1991] distingue entre les objets persistants qui vont bénéficier de la gestion de leurs versions et ceux temporaires qui disparaissent après la session de travail. La relation entre un objet permanent et ses versions est modélisée par un triplet qui relie chaque identificateur d'objet aux trois informations suivantes :

- V : l'ensemble de ses versions; chaque version est représentée par son identificateur unique
- D : un ensemble de paires dont chaque paire correspond à un arc dans l'arbre de versions; la paire (v1, v2) signifie que v1 est dérivée de v2;
- T : un ensemble de paires dont chaque paire associe un identificateur de version à son temps de création.

Toutes ces structures de versions sont équivalentes puisqu'on y trouve toutes les informations nécessaires pour sélectionner une version d'un objet parmi l'ensemble de toutes les versions d'objets, et inversement pour trouver l'objet générique d'une version donnée (par exemple la fonction *objectid* dans [Agrawal et al., 1991] prend comme argument une version et rend son objet générique).

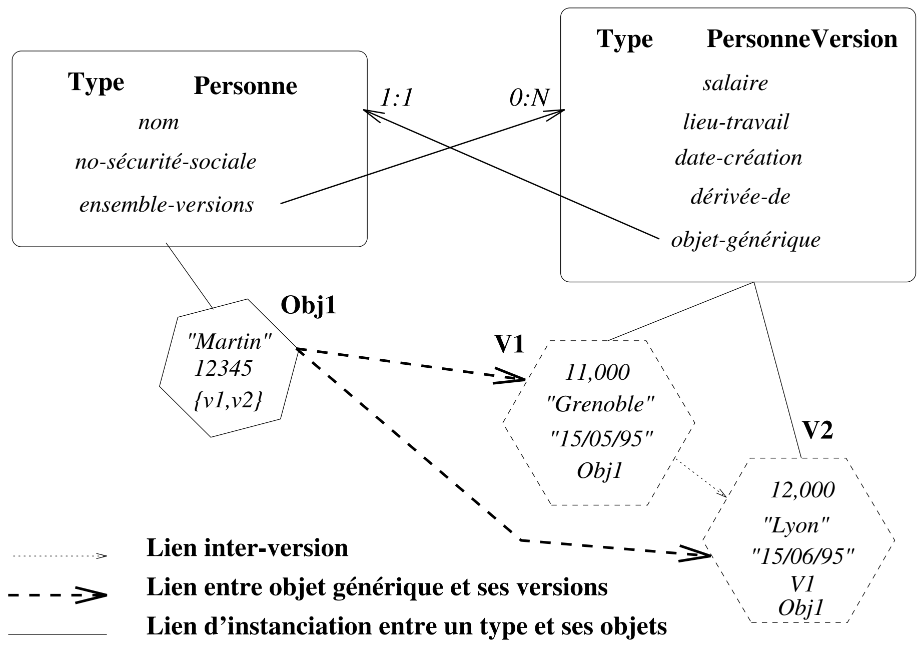


Figure 2.5 - : Un objet générique peut avoir plusieurs versions (ou aucune). Une version appartient exactement à un seul objet générique. L'objet *Obj1* est un objet générique, instance du type générique *Personne*. Il possède deux versions *V1* et *V2*, instances du type *PersonneVersion*.

2.5.3 Identification d'une version d'un objet

L'identification d'une version se fait de manière générale en identifiant en premier l'objet générique puis ensuite la version désirée. Dans la plupart des cas, l'identificateur d'une version (i.e. sa clé) est son numéro dans l'arbre de versions [Dittrich et Lorie, 1988]. Un tel identificateur est local à l'entité à laquelle est associée la version concernée.

Dans les bases historiques, une version est identifiée par sa date de création. Cet identificateur est global car il ne dépend pas de l'entité mais du temps. L'avantage de l'identification globale est le partage implicite de versions [Gançarski, 1994]. S'il n'existe pas une version créée en date t, le système extrait alors la version créée en date t' où t' est la date la plus proche qui succède à t.

Il existe d'autres identificateurs globaux indépendants de la notion du temps. Les approches travaillant avec la notion d'environnement de versions ou de contexte (par exemple [Zdonik, 1986], [Cellary et Jomier, 1990], [Plaice et Wadge, 1993]) utilisent l'idée de l'identificateur global relatif à ce contexte. Le contexte[8] est par définition un regroupement de versions d'objets différents répondant à un critère particulier. Pour la figure 2.6, le contexte *Noir* regroupe les versions de couleur noire des objets de la base. Les versions avec la couleur blanche sont stockées dans le contexte *Blanc*. L'identificateur d'un objet donné dans un contexte précis n'est que l'identificateur d'une version de cet objet relatif à ce contexte. Le couple <Noir, rectangle> (fig. 2.6) est alors la version noire de l'objet rectangle. Pour avoir un triangle blanc il faut donc choisir le contexte Blanc et puis l'objet triangle.

La différence entre tous ces moyens d'identification de versions aura des conséquences directes sur les autres aspects de gestion, particulièrement la gestion des versions d'objets composites étudiée dans la prochaine section.

[8] L'intérêt d'une telle approche est montré plus loin.

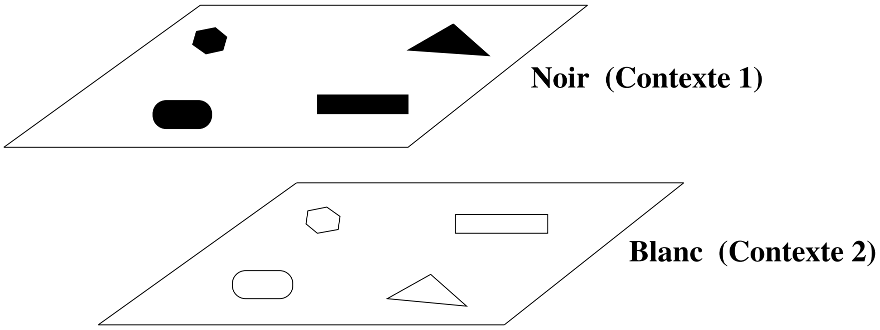


Figure 2.6 - : Un contexte est un regroupement de versions d'objets différents répondant à un critère particulier.

2.6 Modélisation et gestion des versions d'objets composites

2.6.1 Description de la problématique

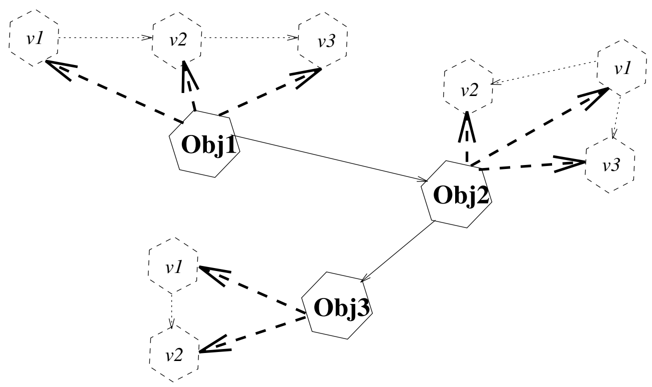
Les objets composites sont ceux qui comportent des références[9] sur d'autres objets (leurs composants), eux-mêmes pouvant à leur tour référencer d'autres objets. Dans les environnements classiques sans versions, lorsqu'un objet Obj est composé d'un objet $Obj1$ et ce dernier a été remplacé par $Obj2$, alors Obj aura un nouveau composant $Obj2$. Mais, avec la présence des versions, le problème est différent et donne place à plusieurs interrogations. Il s'agit d'établir la relation entre les versions d'objets composites (versions composites) et celles de ses composants (versions composantes) (voir fig. 2.7). Cette relation est appelée configuration et la manière de l'établir est appelée *propagation de modification*, ce qui veut dire que toute modification d'une version doit être propagée aux versions qui ont une référence vers elle. Dans cette section, nous étudions les deux types de configuration, *statique* et *dynamique* dépendant respectivement des deux types de propagation, *immédiate* et *paresseuse*.

Il existe deux méthodes de création de versions composites [Talens et al., 1993] : explicite et implicite. Dans le cas d'une création explicite, la version composite est conçue par l'utilisateur de bas en haut en commençant tout d'abord par générer ses versions composantes, ou bien inversement de haut en bas. De toute façon, la relation entre les versions composites et composantes est spécifiée et précisée par l'application. Le problème de configuration se pose plutôt au niveau de la création implicite. La génération implicite d'une version signifie qu'elle n'a été créée que parce que ses (ou une de ses) composantes ont subi un changement. Ainsi, cette génération n'est que le résultat de la propagation des modifications du bas de la hiérarchie de composition vers le haut. Dans la suite, nous détaillons comment cette opération de propagation va contrôler l'évolution d'objets composites.

2.6.2 Références spécifiques et configurations statiques

Le contrôle de la propagation de modifications et la gestion des configurations dépendent du type de la relation composite. Si $Obj1$ est un objet composé de $Obj2$, la relation

[9]Chaque modèle à objets a sa propre sémantique de la référence de composition : *est-partie-de*. Pour nous, il s'agit de la relation de référence, dans son sens le plus large, qui définit des objets référençants (ici appelés composites) et des objets référencés (appelés composants).



la relation de dérivation entre les versions

la relation qui attache un objet à l'ensemble de ses versions

la relation de composition entre les objets

Figure 2.7 - : Objets composites et leurs versions multiples.

entre ces deux objets est dite *spécifique* dans le cas où chaque version de $Obj1$ référence directement une version de $Obj2$. Ce qui signifie que la création d'une nouvelle version de $Obj2$ engendre automatiquement la génération d'une nouvelle version $Obj1$. On parle de propagation *immédiate* de modifications et de configurations *statiques*.

Il n'est pas toujours nécessaire que la mise à jour d'une version référencée entraîne de manière automatique la création d'une nouvelle version de l'objet composite correspondant. La solution est de laisser (de nouveau) le choix à l'utilisateur en déclarant l'attribut de composition comme sensible ou non (c'est le cas du système `Encore` [Zdonik, 1986]) :

- Si l'attribut de composition est déclaré non sensible, cela signifie que sa nouvelle valeur prend la place de l'ancienne dans l'objet composite sans l'incorporation d'une nouvelle version de celui-ci. Ce choix a pour avantage de limiter le nombre de versions créées, qui peut être excessif dans le cas où la profondeur de la hiérarchie de composition est importante. Il faut noter que cette solution n'est pas envisageable dans les bases historiques à cause des pertes d'informations concernant les anciennes versions composantes.

- Si l'attribut de composition est déclaré comme sensible alors l'objet composite aura une nouvelle version créée automatiquement chaque fois que la version composante, référencée par l'attribut sensible, est générée. Ce choix représente une solution inverse par rapport à la précédente dans le sens où toute modification est conservée. Toutefois le nombre de versions de chaque objet composite devient très élevé, et cette solution risque d'être coûteuse.

La figure 2.8 est un exemple de la propagation *sélective* de modification. Une version de $Obj4$ est dérivée. Quelles sont alors les configurations possibles? Trois d'entre elles sont données dans la partie (b). La première configuration est envisagée lorsque l'attribut D de $Obj3$ est sensible (création de la version $Obj3.V1$), l'attribut C de $Obj2$ n'est pas sensible ($Obj2$ ne possède toujours pas de versions), l'attribut B de $Obj1$ est sensible (création de la version $Obj1.V1$ à cause de la création de la version composante $Obj3.V1$). La deuxième configuration correspond à la situation suivante : C est sensible, D est non sensible et A est sensible. Quant à la troisième configuration, elle est possible lorsque C, D et A sont sensibles. Pour ce dernier cas, le fait que $Obj1$ possède un attribut non sensible (l'attribut

B) n'a pas empêché la création d'une version de ce objet. Ceci est dû à l'objet partagé ($Obj4$) et à la sensibilité des deux attributs qui le référencent (C et D). Par rapport à cette situation, l'avantage qu'offre les attributs non sensibles (le contrôle du nombre de nouvelles versions créées) ne reste plus valable.

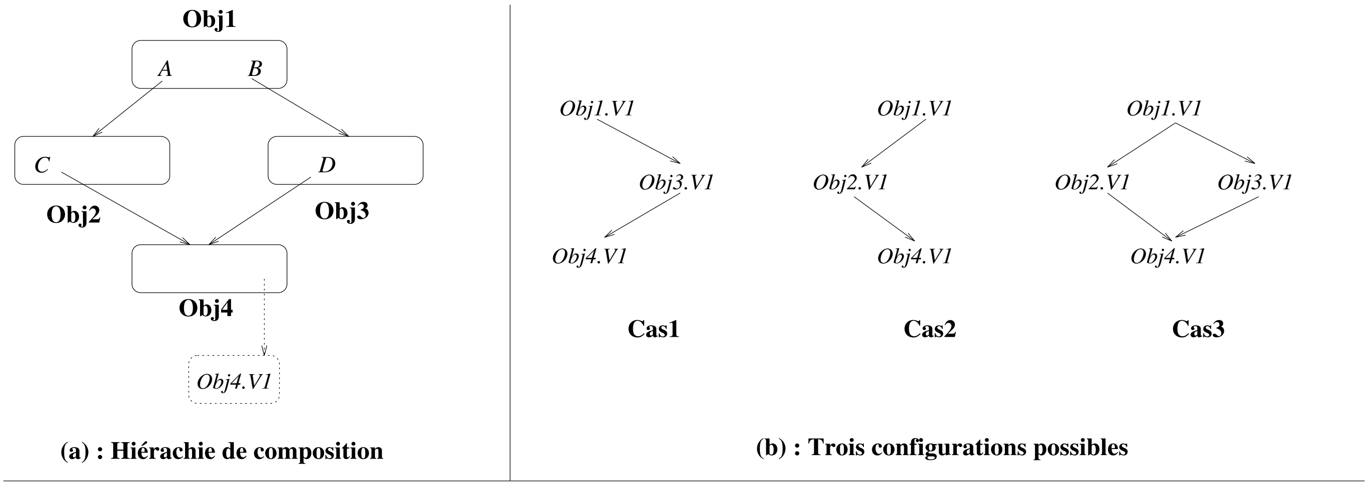


Figure 2.8 - : Différentes configurations dues à un choix différent de la sensibilité des attributs composites.

Une autre sorte de configuration appelée *dynamique* a été proposée qui remplace les références spécifiques par des références *dynamiques*, et qui est basée sur le principe de la propagation *paresseuse* des modifications. Nous montrons comment cette technique va diminuer le nombre de versions composites créées.

2.6.3 Références génériques et configurations dynamiques

Contrairement aux références spécifiques (qui pointent directement sur une version précise de l'ensemble des versions d'un objet), les références *génériques* pointent sur l'objet générique, qui représente dans ce cas l'interface de ses versions pour les autres objets du modèle. La valeur d'une référence spécifique (qui est la version sur laquelle elle pointe) est connue à tout moment. En revanche, une référence générique n'a pas de valeur précise (puisqu'elle pointe sur l'ensemble des versions d'un objet générique). Sa valeur sera calculée dynamiquement lorsque c'est nécessaire (on dit que les configurations sont construites dynamiquement [Sciore, 1994]).

Il existe plusieurs méthodes afin de spécifier une référence générique (i.e. rendre spécifique une référence générique). Dans le système `Ode` [Agrawal et al., 1991] toutes les références génériques ont une valeur par défaut : la dernière version créée (considérée comme la version courante). La configuration courante est donc toujours prête à être utilisée, par contre il faut calculer les autres. `Orion` [Kim et al., 1990] offre des opérations permettant de passer d'une référence dynamique à une référence statique et cela en précisant le numéro de la version voulue (Si A est un attribut composite et si A est une référence dynamique alors $A.n$ réfère la version composante numéro n). Cette solution est plus générale que celle de `Ode` mais elle reste limitée dans le sens où un utilisateur ne peut pas, par exemple, demander de configurer l'objet dont les versions composantes satisfont une condition particulière. Ceci est par contre possible dans les bases historiques où les conditions particulières sont des dates données. Chaque version dans une base historique étant étiquetée par sa date de création, le système va pouvoir facilement identifier les versions désirées et les affecter comme valeurs des références génériques. [Sciore, 1991] propose une méthode plus générale permettant d'associer à n'importe quelle référence générique un prédicat de sélection arbitraire (y compris des prédicats temporels).

Sciore définit les "view objects" comme des objets génériques dont l'ensemble des versions est créé dynamiquement lors de la demande. Si A est une référence à un objet générique, et exp est une condition de sélection alors $A[exp]$ est une référence à un "view object" ou plus exactement à la version visible de "view object" satisfaisant exp. Cette version est donc la vue de la référence A selon exp. En effet, les "view objects" sont des objets "virtuels" (ils n'existent pas physiquement) associés à des objets génériques. Chaque objet générique peut avoir autant de "view objects" que de configurations désirées par l'utilisateur. Les configurations sont ainsi trouvées dynamiquement en répondant aux requêtes de ce dernier. Le gestionnaire `Adèle` permet également de reconstituer dynamiquement une configuration à partir de certaines versions de composants logiciels sélectionnées par des prédicats offerts par le système [Estublier et Casallas, 1994]. En pratique, bien que ces approches permettent une plus grande souplesse dans la détermination des configurations, elles dépendent fortement de l'expressivité du langage de requêtes de chaque modèle. Si le langage ne permet pas d'exprimer de telles expressions de sélection, la méthode de Sciore (ou de `Adèle`) ne peut pas, par conséquent, être appliquée.

Pour conclure, les références génériques sont plus adéquates que les références statiques. Elles sont un moyen d'obtenir des informations sur tout l'historique de l'objet générique référencé (puisqu'elles pointent sur l'ensemble de ses versions). De plus, le système n'a pas besoin de créer de nouvelles versions composites chaque fois qu'un composant a été modifié, mais plutôt chaque fois que l'utilisateur demande l'accès à l'objet composite concerné. Néanmoins, il est souhaitable d'avoir la possibilité d'utiliser les références statiques et dynamiques dans les objets composites (comme le proposent [Kim et al., 1987] et [Agrawal et al., 1991]). Lorsqu'un objet a une seule version, il est plus efficace et plus rapide de référencer directement cette version unique (i.e. d'utiliser une référence statique).

Dans la section suivante, nous allons introduire les *contextes* qui sont aussi des mécanismes de configuration dynamique. L'idée est assez similaire à celle des vues[10] en général (et celles de Sciore en particulier) mais le point fort de cette approche est qu'il n'y pas de calcul de référence à faire. Le contexte ne contient que des configurations correctes prêtes à l'utilisation.

2.6.4 Notion de contexte

Dans tout ce qui a été vu jusqu'à maintenant, les versions multiples d'un objet existaient simultanément dans la même base. Cette existence en même temps et dans le même endroit a comme première conséquence de mettre en cause la consistance de la base de données. Les configurations sont une manière de rassembler les versions compatibles entre elles d'un même objet composite. Toutefois, elles sont des solutions locales (cohérence d'un objet individuel et non pas de l'ensemble des objets d'une base). Elles exigent d'introduire de nouvelles fonctionnalités de sélection pour accéder séparément à une seule version d'un objet à la fois et enfin, c'est l'utilisateur qui a la charge de les définir et de les gérer.

Nous allons nous intéresser à une autre technique (celle des contextes) permettant de séparer explicitement, dès le début, les différentes versions d'un objet. Un contexte est

[10] Une synthèse de principaux travaux sur les vues est exposée dans [Al-Jadir et al., 1993].

par définition un regroupement de versions cohérentes, chacune correspond à un objet différent de la base, répondant à un critère particulier. L'utilisateur de `Encore` [Zdonik, 1986] définit un contexte en indexant une suite continue de transparents[11]. Un transparent regroupe des versions produites de la même transaction (ensemble d'opérations de modification). Pour `PIE` [Goldstein et Bobrow, 1980] l'utilisateur définit un contexte de travail en déterminant un ordre quelconque sur les couches[12]. Les versions qui doivent être utilisées ensemble sont placées dans la même couche. Ces contextes, une fois définis, vont être créés dynamiquement par le système et ne peuvent plus être modifiés. Ils permettent donc de préserver l'historique des changements effectués.

Lors de la sélection d'un contexte (le contexte est dit *activé*), toutes les versions qui lui appartiennent sont sélectionnées et les références entre versions y sont résolues. Ce qui revient à dire qu'une version composite dans un contexte C est constituée de ses versions composantes stockées dans le même contexte C. Ainsi, un contexte n'est qu'une configuration possible de tous les versions composites qu'il contient. Un autre avantage réside dans le passage facile d'une configuration d'objets à une autre; il suffit de sélectionner le contexte concerné (en changeant l'ordre des couches ou en choisissant une autre séquence de transparents).

Cette notion de contexte est considérée comme une extension de la notion de référence générique [Chou et Kim, 1986]. Elle permet de réaliser des configurations dynamiques mais de manière beaucoup plus simple pour le système, et moins compliquée du point de vue de l'utilisateur, qui est libéré de cette tâche.

Ces arguments vont nous conduire, comme nous le verrons au § 3.3, à adopter cette technique et à manipuler des contextes multiples des entités mono-version (avec une seule version par contexte) au lieu de gérer un seul contexte avec des objets multi-versions (avec plusieurs versions à la fois). Les contextes, dans notre cas, sont représentés par des versions de base de connaissances. Chaque version de base contient les versions de connaissances qui sont cohérentes entre elles. Cette idée est proche de celle proposée dans [Cellary et Jomier, 1990]. Pourtant, il y a de nombreuses différences qui seront soulignées dans 3.7 en comparant les deux méthodes.

Après avoir discuté les différentes approches de la gestion des versions des objets, nous allons maintenant traiter l'évolution des types de ces objets et étudier les solutions proposées pour la gestion de leurs versions.

2.7 Évolution des types

L'évolution d'un type dans un modèle à objets est exprimée par l'addition, la modification ou la suppression d'attributs dans sa description ou d'opérations dans son interface. Dans un modèle sans le support de versions des types, la nouvelle représentation du type remplace la représentation précédente. Parallèlement, toute la base de données doit être réorganisée afin de pouvoir réutiliser les anciens objets du type modifié. Traditionnellement, ces objets vont être mis à jour afin d'être adaptés à la nouvelle représentation de leur type [Odberg, 1992].

Tenant compte de nos besoins, la solution que nous allons étudier pour gérer l'évolution des types est celle du versionnement. Cette solution permet de garder l'accès aux

[11] Appelés "slides".
[12] Appelées "layers".

données dont on a modifié le type. Ceci est très important dans les environnements où plusieurs personnes travaillent de manière collective et partagent le développement de la même entité (ce qui est notre cas). Si un utilisateur a accès à un type T à travers ses objets, alors un autre utilisateur pourra modifier T en une nouvelle version T' pendant que le premier programmeur s'intéresse toujours aux anciens objets.

Les aspects de l'évolution des structures de données, que nous allons étudier dans les sections suivantes, concernent la gestion des versions des types modifiés et la gestion de leurs objets.

2.7.1 Gestion des versions des types

Il s'agit de contrôler la coexistence de plusieurs versions des types. Dans un premier temps, nous allons décrire la relation entre les différentes versions au sein d'une hiérarchie de types (i.e. relation entre une version de type et les versions de ses sous-types). Nous montrons ensuite la relation entre les différentes versions au sein du même type.

Versions des types et des sous-types

Ce paragraphe étudie l'impact de la création d'une version d'un type sur son graphe de spécialisation (ses sous-types). Dans `Encore` [Zdonik, 1986], la granularité de version choisie est le type. Chaque modification d'un type crée automatiquement une nouvelle version de celui-ci et de chacun de ses sous-types. En ce qui concerne `Avance`, les auteurs [Björnerstedt et Britts, 1988] adoptent la même technique mais ils constatent que les changements qui sont internes à un type et qui n'ont pas d'effets sur les autres types ne nécessitent pas la création de nouvelles versions de ces derniers (par exemple la modification des réalisations des méthodes d'un type). Afin de définir un état complet de la base, l'utilisateur doit choisir une version particulière pour chaque type faisant partie de cet état. `Orion` [Kim et al., 1989] contrôlent les versions de type au niveau du schéma[13]. Chaque modification dans un type a pour conséquence la génération d'une nouvelle version complète du schéma qui le contient. Le modèle `Farandole2` ([Falquet, 1990] [Andany et al., 1991]) a choisi une granularité intermédiaire par rapport aux deux ci-dessus qui sont décrites comme trop étroite (cas d'`Encore`) ou trop large (cas d'`Orion`). Les versions sont supportées au niveau des vues. Une vue (ou un contexte) est une portion du schéma regroupant un certain nombre de types selon une sémantique précise. Lorsqu'un type évolue au sein d'une vue, une nouvelle version de celle-ci est créée.

Pour conclure, afin d'assurer la cohérence de la base de données avant et après toute modification, toutes ces approches définissent des invariants. Ceux-ci, appelés *invariants d'un type*, sont des contraintes d'intégrité qui doivent être maintenues à travers le changement d'un type [Casais, 1990]. Les invariants de sous-typage [Ahmed-Nacer, 1994] imposent aux versions d'un type modifié de respecter ses relations avec ses super-types et ses sous-types.

Graphe de versions du même type

La gestion de la relation entre plusieurs versions d'un type n'est pas la même dans tous les systèmes. On retrouve la notion de *version-type-générique*, similaire à la notion

[13]Un schéma est une hiérarchie de types dans les bases de données.

d'objet générique, représentant l'ensemble des versions d'un type [Talens et al., 1993]. Les relations de dérivation entre les versions de types permettent dans cette approche de spécifier les attributs à ajouter, à supprimer ou à modifier entre deux versions successives. De même, pour le système `Farandole2` chaque vue définie sur la base est associée à une vue générique qui rassemble ses propres versions.

Le système `Encore` introduit le concept de "Version Set Interface" (VSI) qui est un type virtuel contenant l'union de tous les attributs des versions du type correspondant. La création d'une nouvelle version de celui-ci implique l'extension des attributs du type VSI afin de prendre en compte les attributs contenus dans la version créée et qui ne sont pas présents dans les autres versions. Dans le même esprit, Odberg [Odberg, 1992] associe à chaque type, un type appelé "Union Set Type Version" (USTV) qui est défini implicitement comme un sous-type de chacune des versions du type concerné. Cela signifie que ce nouveau type va hériter les propriétés et les méthodes de ces versions.

Tous ces mécanismes ont été mis au point afin, d'une part de gérer la hiérarchie de dérivation des versions d'un type et, d'autre part de faciliter la manipulation des objets des types modifiés. Ceci est détaillé dans la section qui suit.

2.7.2 Gestion des objets des types modifiés

Les modifications effectuées sur un type doivent être répercutées sur ses objets déjà instanciés [Nguyen et Rieu, 1989]. Quelle que soit la mesure prise, elle doit garantir que les objets satisfont la description de leur type d'appartenance. La littérature contient deux techniques différentes pour obtenir ce résultat : la conversion et le filtrage.

Conversion

Lorsqu'un type est modifié, tous ses objets vont également être mis à jour conformément à la nouvelle définition de ce type. Cette opération de conversion implique la transformation des valeurs des anciens objets afin de les attacher physiquement au nouveau type (i.e. les détacher de leur ancien type). Le versionnement dans ce cas n'est pas nécessaire puisque les applications ne peuvent plus accéder à la définition précédente du type modifié.

Plusieurs systèmes ont adopté cette technique avec les nuances concernant le moment où est réalisée la conversion. `GemStone` [Penny et Stein, 1987], par exemple, convertit les objets dès l'affectation des changements sur son type. C'est le principe de la *conversion immédiate*, tandis que la *conversion paresseuse*, utilisée par `Avance` et `Orion`, consiste à ne modifier les objets que lorsqu'on y accède pour la première fois après la modification de leur type. Le système `O2` [Zicari, 1991] a exploré les deux techniques de conversion.

Filtrage

L'autre mécanisme est celui du filtrage. Il permet aux objets créés selon une version d'un type d'être vus comme s'ils étaient créés selon une version différente de ce type et cela en définissant des filtres sur la base d'objets.

Un *filtre* est une interface d'encapsulation définie entre une application et tous les objets d'un type. Ces objets ne sont visibles qu'à travers leurs filtres associés (voir fig. 2.9). Ainsi, un filtre intercepte tous les messages envoyés à ses objets. Il va ensuite les traiter et envoyer ou bien la version d'objet désirée ou bien lever une exception lorsque l'application fait référence à des propriétés qui n'existent plus. Selon cette méthode, plusieurs versions d'un type sont gérées et leurs anciens objets sont préservés.

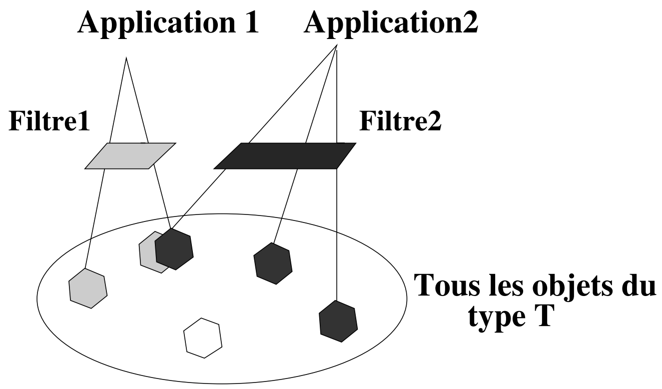


Figure 2.9 - : Tous les objets du type T persistent indépendamment des versions de T. Ce n'est que lorsqu'ils sont accessibles via un filtre qu'ils vont être instanciés d'une version particulière de T, indiquée par l'application utilisant le filtre.

Le filtrage a été implémenté de différentes façons. Pour [Ahmed-Nacer, 1994], les objets d'un type sont aussi des objets de ses versions. Chaque version de type est définie comme un point de vue sur l'ensemble des objets de son type, en filtrant les attributs et les propriétés faisant partie de sa définition. La figure 2.10 explique l'idée. Obj est un objet de type T qui possède deux versions $V1$ et $V2$. Il existe donc deux points de vue de l'objet Obj. Celui-ci est visible selon le premier point de vue lorsqu'il est manipulé à travers la première version ($V1$) de son type. L'utilisateur peut accéder aux attributs a et b. Par contre, pour accéder à la valeur de l'attribut d, l'utilisateur doit travailler avec le deuxième point de vue en manipulant Obj selon la version $V2$.

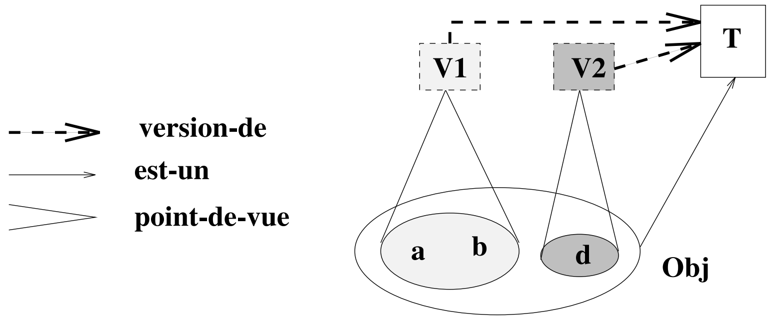


Figure 2.10 - : La version de type est considérée comme un point de vue sur les objets de type. Ceci assure qu'un même objet peut être accédé à la fois par plusieurs versions de son type.

Dans [Odberg, 1992], le *"Union Set Type Version"* d'un type T contient implicitement tous les objets de ce dernier. De cette manière, tous les objets héritent logiquement de la totalité des propriétés définies dans toutes les versions de leur type. Ils peuvent donc répondre aux interrogations d'une application concernant n'importe quelle version de T. En effet, l'objectif d'Odberg est de rendre transparent tout changement d'un type vis-à-vis des objets déjà existants et des applications. Cette transparence est manifestée par le fait que tous les objets sont accessibles indépendamment de leurs versions de type. En ce qui concerne `Encore`, tous les objets d'un type le sont aussi pour le *"Version Set Interface"* (VSI) associé à ce type. Ceci rend possible la conservation des anciens objets même si leur type a été modifié. Ce système offre des exceptions[14] qui sont activées avant ou après un accès échoué à l'attribut auquel elles sont associées. Le rôle de ces exceptions consiste à renvoyer une valeur par défaut de l'attribut concerné. Supposons que le type T ait un seul attribut a. Un objet Obj a été créé selon cette définition. Pour une certaine raison, T a été modifié en lui ajoutant l'attribut b. Les interrogations concernant T vont passer via son interface VSI (qui possède les deux attributs a et b). Si un utilisateur demande la valeur de b dans Obj, l'exception associée à b dans la deuxième version va être appelée et une valeur par défaut est renvoyée comme réponse.

Conclusion sur la gestion des objets des types modifiés

Convertir toutes les objets dont le type a été modifié semble être l'approche la plus naturelle afin de traiter la propagation des changements des structures de données. Toutefois, la mise en œuvre de cette conversion (immédiate ou paresseuse) nécessite des processus de calcul coûteux et d'autre part implique une perte d'information concernant les anciens objets transformés. Le principe de l'autre possibilité, le filtrage, consiste à assurer une sorte de compatibilité entre les anciennes et les nouvelles représentations des objets (de cette manière les anciens objets peuvent persister). L'inconvénient de cette approche réside dans son coût et sa complexité élevés. D'une part, l'utilisation des traitements d'exception exige des modifications dans toutes les versions du même type (lorsqu'un attribut est ajouté ou supprimé d'un type, des exceptions doivent être introduites dans toutes ses autres versions). D'autre part la transparence des changements nécessite le développement d'une série de fonctions substituant une description d'un objet à n'importe quelle autre description à chaque demande d'accès.

Pour conclure, il est clair que pour pouvoir gérer plusieurs versions d'un type telles que les anciens objets et leurs anciennes descriptions cohabitent avec les nouveaux objets et leurs nouvelles descriptions, il faut écarter la technique de conversion. Le choix (qui reste) est de suivre le principe du filtrage en essayant de l'améliorer ou bien de proposer une troisième et nouvelle approche.

2.8 Synthèse et conclusion

Dans ce chapitre, nous avons étudié le versionnement des objets et des types. Pour chacun, nous avons expliqué la problématique et identifié ses différents aspects. À l'issue de cette étude, nous dégageons les constatations suivantes :

1. les versions sont indispensables afin de préserver les changements effectués dans une base. Cependant, des moyens additionnels sont nécessaires afin de permettre la gestion (la représentation, l'accès, etc.) de la coexistence de plusieurs versions des entités (i.e. les objets ou les types).

2. les versions d'une entité doivent être organisées en un graphe afin de pouvoir travailler avec des versions variantes. La plupart des bases de données temporelles (à l'exception des travaux tels que [Wuu et Dayal, 1993] et [Sciore, 1994]) ne permettent qu'une évolution linéaire des versions puisqu'elles sont ordonnées totalement en fonction du temps.

[14] Appelées "handlers".

3. l'utilisation des versions remet en cause la cohérence de la base. Il s'agit du problème de la configuration lors du versionnement des objets complexes avec des références à des autres objets. Lors de versionnement des types, le problème concerne la cohérence du graphe de spécialisation et celle des objets du type modifié.

Après avoir examiné et analysé les systèmes existants dans la littérature, la plupart des systèmes ont dédié leur gestionnaire de versions à un modèle de données particulier. Cette dépendance a de lourdes conséquences sur le modèle lui-même et sur la mise en œuvre de ces systèmes. En même temps, ces derniers souffrent des insuffisances et des inconvénients suivants :

- afin de gérer l'ensemble des versions et leur relation de dérivation, une grande partie des systèmes surcharge leur modèle de données en introduisant de nouvelles notions comme *objet générique et type générique*. Même si la plupart du temps c'est aux systèmes de gérer ces notions, dans certains cas (l'utilisation des références dynamiques), le programmeur est obligé de les utiliser.

- dans la continuité de l'argument précédent, les utilisateurs bénéficiant du versionnement des entités conçues doivent travailler avec des modèles de données compliqués. Premièrement, ils doivent naviguer dans deux hiérarchies interconnectées : la hiérarchie d'héritage traditionnelle et la hiérarchie de dérivation de versions. Ils doivent avoir une grande maîtrise des principes mis en œuvre afin d'exploiter correctement les versions de leurs données. Deuxièmement, ils doivent prendre en considération de nouveaux critères ajoutés à ceux de conception (distinguer entre sensible et non, entre objets qui peuvent avoir des versions et ceux qui n'ont pas, etc.).

- peu de systèmes offrent la possibilité de gérer les versions de types et d'objets à la fois. Néanmoins, même pour ceux qui le permettent, on constate qu'il n'existe pas une solution générale traitant uniformément ces deux genres de versionnement. De plus, chaque approche définit ses propres termes, dont le sens est différent d'un système à un autre (par exemple vue d'objets, vue de schémas, contexte). Cette limite risque de diminuer les avantages des gestionnaires de versions et de compliquer une fois de plus leur réalisation et leur exploitation.

- la gestion de la cohérence est un problème crucial. Toutefois, les réponses apportées à cette question ne sont pas toujours efficaces. Hormis les approches basées sur le contexte, et qui manipulent les versions de différents objets qui vont ensemble, les autres approches consistent à définir des configurations. Celles-ci représentent un moyen de regroupement des objets qui se référencent. Cependant leur reconstitution n'est pas toujours simple et leur consistance doit être vérifiée à chaque fois.

Notre objectif est de gérer des versions complètes des bases de connaissances. Nous montrons dans les chapitres suivants comment nous avons pu répondre à cet objectif tout en essayant de prendre en compte les apports constatés et de pallier les inconvénients soulignés ci-dessus.

Chapitre 3

Le système de gestion de versions des bases de connaissances

Avant de présenter le système de gestion des versions, nous nous intéressons tout d'abord à la sémantique des versions (section 3.1). Nous donnons ainsi la définition d'une version dans notre système. La deuxième section (section 3.2) détaille le formalisme utilisé afin de représenter les connaissances au sein de la base. Malgré la généralité de notre modèle de versions, nous tenons compte de ce formalisme[1] pour préciser le sens du mot connaissance. Nous détaillons ensuite le modèle que nous avons conçu afin de représenter les versions. Nous nous intéressons aux différentes opérations mises à la disposition de l'utilisateur (section 3.4) ainsi qu'à leurs répercussions sur l'état de la base (section 3.5). La section 3.6 présente le bilan concernant notre système de versions. Ce bilan est étudié par rapport aux objectifs que nous avons fixés et par rapport aux limites des autres travaux dans la littérature signalées dans le chapitre précédent. Enfin, la section 3.7 compare notre approche avec une autre approche similaire à la nôtre, celle de Cellary et Jomier [Cellary et Jomier, 1994].

3.1 Les versions et leur sémantique

La progression scientifique et donc l'évolution des connaissances du domaine concerné est permanente. Ceci a pour conséquence directe la variation avec le temps de la base contenant les connaissances des chercheurs. Lorsqu'une connaissance est modifiée, son ancienne valeur est remplacée par la nouvelle. De même, la suppression d'une connaissance implique la perte de ses traces. Notre objectif est de garder l'historique de tous les changements effectués sur une base de connaissances (BC). Pour ce faire, nous allons gérer les différentes versions de celle-ci.

Par définition, une version d'une base de connaissances reflète un état (à un moment donné) de l'évolution de cette base au cours du temps. Nous définissons un état comme l'ensemble des structures et des valeurs des connaissances de la base. On s'appuie donc sur une sémantique temporelle[2] au même titre que les bases de données temporelles [Klopprogge et Lockemann, 1983]. Par contre, nous ne faisons pas de différence entre un temps

[1] Le formalisme choisi pour représenter les connaissances est pris comme exemple. Il peut être remplacé par n'importe quel autre formalisme à base d'objets.

[2] Nous rappelons qu'une sémantique temporelle signifie que chaque fois que la base est modifiée, une nouvelle version de celle-ci est créée.

de transaction[3] (appelé temps physique) et un temps de validité[4] (appelé temps logique) [Gadia, 1993] car les deux sont identiques dans notre cas. Ceci fait partie de nos choix. Nous considérons que l'instant de l'insertion d'une connaissance dans une base représente le moment à partir duquel celle-ci est considérée comme valide dans le monde réel. Dès qu'une connaissance est acquise par un chercheur, elle sera intégrée dans la BC de ce dernier.

Un autre point de divergence avec l'approche des bases de données temporelles est que notre modèle ne réserve pas un traitement spécial pour le temps. Son rôle dans nos requêtes est de spécifier la version la plus récente, la plus ancienne, ou encore celle créée à un moment donné. Ainsi, nous utilisons le temps pour identifier une entité recherchée, mais notre but n'est pas de manipuler le temps (durée, intervalle, période, etc.) comme c'est le cas pour les BD temporelles [Adiba et al., 1987].

Contrairement aux bases temporelles, les versions d'une base selon notre approche sont ordonnées totalement en fonction du temps et partiellement en fonction des relations de prédécesseur-successeur définies entre elles.

3.2 Modèle choisi pour la représentation de connaissances

Bien que les deux modèles de représentation de connaissances et de versions soient indépendants, nous allons décrire le premier afin de préciser le sens du mot connaissance utilisé auparavant et le remplacer par ses expressions symboliques dans le modèle choisi. L'objectif de la description du modèle est de fournir un aperçu de tous les aspects du modèle que nous devons prendre en considération pour l'élaboration du système de gestion de versions.

3.2.1 Description générale

Le modèle de représentation de connaissances choisi est celui de `Shirka`. `Shirka` est un système de gestion de bases de connaissances réalisé par François Rechenmann [Rechenmann, 1988]. Il s'inscrit dans le cadre des modèles centrés-objets. Nous y retrouvons les deux notions essentielles : classes et instances. Celles-ci sont décrites par une seule entité, appelée *schéma*.

Un *schéma de classe* repose sur la structure : schéma-attribut-facette (fig. 3.1). Le premier élément est le nom du schéma servant à l'identifier de manière unique. Il est suivi d'une liste d'attributs. Chaque attribut est défini par son nom et possède un type, introduit par l'une des deux facettes *$un* ou *$liste-de*. Le domaine de valeurs de types peut être restreint grâce aux facettes *$intervalle* et *$domaine*. Les valeurs de types sont des schémas ou des références à d'autres schémas.

La figure 3.1 décrit le schéma de la classe **Personne** comportant trois attributs : *prénom*, *âge* et *date-naissance*. L'attribut *âge* est défini par deux facettes *$un* et *$intervalle*. La facette *$un* de l'attribut *date-naissance* est une référence à un autre schéma **Date** (voir fig. 3.2).

Les instances sont des objets particuliers, représentants des classes. Une instance est décrite par un schéma. Un *schéma d'instance* contient les valeurs des attributs définis dans

[3] Un temps de transaction est le moment de l'enregistrement d'un événement dans la base.

[4] Un temps de validité est le moment où l'événement se déroule dans le monde réel.

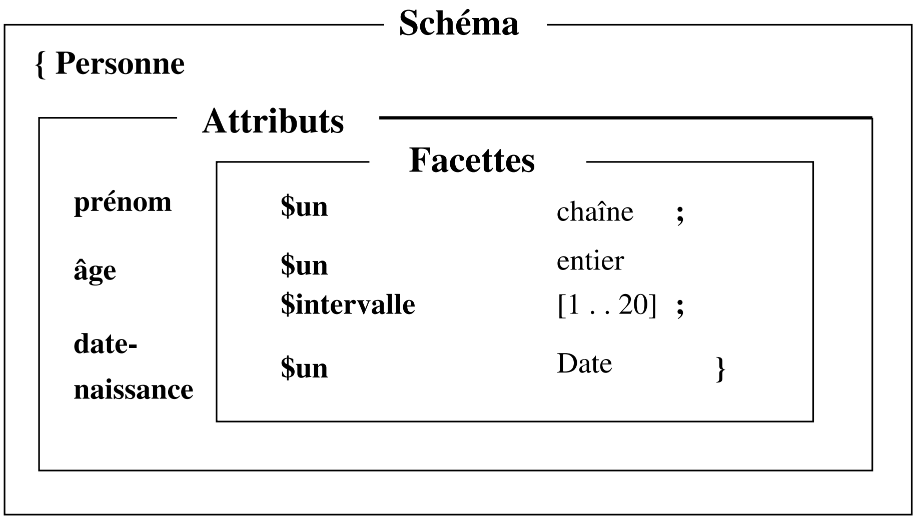


Figure 3.1 - : Structure d'une classe **Personne** décrite par un schéma.

{ Date			
jour	**$un**	entier	
	$intervalle	[1 . . 31]	**;**
mois	**$un**	chaîne	
	$domaine	"janvier" "février" "mars" "avril" " mai" "juin" "juillet" "août" "septembre" "octobre" "novembre" "décembre"	**;**
année	**$un**	entier **;** **}**	

Figure 3.2 - : Schéma de la classe **Date**.

sa classe. Dans `Shirka`, le lien d'instanciation est caractérisé par l'attribut *est-un* (voir fig. 3.3). La figure 3.3 montre un schéma de l'instance *P0*. La valeur de l'attribut *date-naissance* est, dans ce cas, un schéma d'une instance de la classe **Date**. Cette instance est non nommée et donc non partageable et elle existe tant que l'instance *P0* existe. La figure 3.4 montre un autre schéma de l'instance *P0* où la valeur de l'attribut *date-naissance* est une référence à l'instance *D0* de la classe **Date**.

```
{ P0
    est-un          =   Personne     ;
    prénom          =   " Martin "   ;
    âge             =   20           ;
    date-naissance  =   { jour   =  27         ;
                          mois   =  "octobre"  ;
                          année  =  1975       }
```

Figure 3.3 - : Définition du schéma de l'instance *P0* de la classe **Personne**. L'attribut *date-naissance* est directement une instance de la classe **Date**.

```
{ P0                                    { D0
  est-un          =  Personne   ;         est-un  =  Date       ;
  prénom          =  " Martin " ;         jour    =  27         ;
  âge             =  20         ;         mois    =  "octobre"  ;
  date-naissance  =  D0         }         année   =  1975       }
```

Figure 3.4 - : Définition du schéma de l'instance *P0* de la classe **Personne**. L'attribut *date-naissance* est une référence à l'instance *D0* de la classe **Date**.

Les classes sont organisées en une hiérarchie à racine unique, la classe **Objet**, définie par `Shirka`. Les informations les plus générales sont mises en commun dans des classes, dites sur-classes, à partir desquelles sont créées des sous-classes contenant des informations de plus en plus spécifiques. Être plus spécifique signifie contenir une connaissance plus précise ou bien contenir une connaissance complémentaire (voir fig. 3.5). Cette hiérarchie de classes est appelée un *graphe de spécialisation*. L'attribut *sorte-de* dans la description d'une classe permet de la déclarer comme une spécialisation (ou une sous-classe) d'une autre classe. Dans la figure 3.5 la classe **Etudiant-diplômé**, qui est une sous-classe de **Etudiant**, enrichit sa sur-classe en ajoutant les deux attributs *diplôme* et *date-diplômes*. Tandis que la définition de l'attribut *âge* de la classe **Etudiant** complète celle du même attribut dans sa sur-classe **Personne**.

Il est à signaler que pour `Shirka` l'ensemble des instances d'une classe est inclus dans les ensembles des instances de ses sur-classes. Dorénavant nous appellerons cette propriété *l'inclusion des instances*. Ainsi, une instance de la classe **Etudiant** est également une instance de la classe **Personne**.

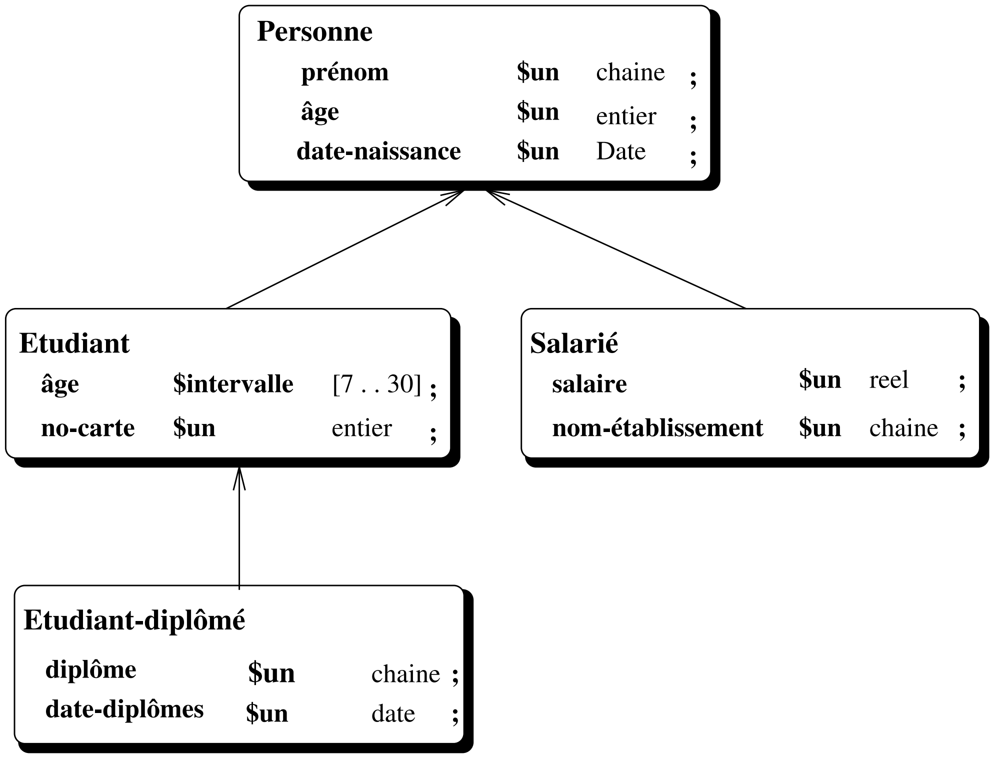


Figure 3.5 - : Un graphe de spécialisation dans `Shirka`.

`Remarque :` `Shirka` maintient les informations concernant la hiérarchie et les instances d'une classe au sein de celle-ci. Chaque classe contient le nom de sa sur-classe directe (via l'attribut *sorte-de* exprimé explicitement dans la description de la classe), les noms de ses sous-classes directes (via l'attribut *spec* ajouté par le système à la représentation interne de la classe) et les noms de ses propres instances (via l'attribut *inst* ajouté par le système à la représentation interne de la classe). Ceci signifie que lorsque par exemple une instance est créée, la liste des instances de sa classe d'appartenance doit être mise à jour par le système. Cette caractéristique sera utilisée lors de la gestion de la cohérence d'une version de la base (Cf. § 3.5).

La conception d'une base utilisant `Shirka` consiste à définir les classes avec les attributs qui leur sont attachés, à construire la hiérarchie des classes et à créer les instances de ces classes. Ces instances peuvent ne pas être complètes (i.e. ne pas avoir des valeurs pour tous leurs attributs). Un des intérêts de `Shirka` est de pouvoir inférer les valeurs inconnues en utilisant les connaissances déjà existantes dans la base. Nous abordons par la suite les trois moyens d'inférence proposés : attachement procédural, filtrage et classification. Nous examinons rapidement les aspects qui nous apparaissent utiles dans notre cas.

3.2.2 Moyens d'inférence

Attachement procédural

L'*attachement procédural* constitue un des moyens d'inférence des valeurs d'attributs indéterminés et un des mécanismes de raisonnement [Mariño, 1993]. Le principe général consiste à associer une procédure à une donnée, cette procédure étant activée lors des accès à la donnée. Dans notre cas, la donnée est un attribut de classe et c'est grâce à des facettes spéciales[5] que l'attachement procédural est mis en œuvre (voir fig. 3.6). La procédure est activée lors de la consultation ou de la mise à jour d'une instance.

```
{ Personne
      sorte-de            =             Objet       ;
      nom                 $un           chaine
      âge                 $un           entier      ;
                          $sib-exec  {  Différence
                                           op1      $valeur   1995            ;
                                           op2      $var <-   date-naissance  ;
                                           diff     $var ->   âge             }
      date-naissance      $un           Date        }

{ Différence
     sorte-de             =             Méthode      ;
     nom-fct              $valeur       différence   ;
     op1                  $un           entier       ;
     op2                  $un           entier       ;
     diff                 $un           entier       }

(de différence  (instance)
        (affect  'diff  (-  (val?  'op1)     (val? 'op2))))
```

Figure 3.6 - : Exemple d'attachement procédural pour calculer l'âge d'une personne à partir de sa date de naissance.

Selon la figure 3.6, l'âge d'une personne peut être calculé, s'il est inconnu, par le moteur d'inférence qui déclenchera la facette *$sib-exec* dans le schéma de la classe **Personne**. La procédure de calcul est *compter-années*, définie par un schéma de classe[6] de `Shirka`, est une fonction `Le-Lisp` dont l'unique argument est une instance de la classe **Date**. La facette *$var* $\rightarrow$ spécifie l'attribut de la procédure dont la valeur sera rendue comme résultat. Nous n'expliquons pas dans le cadre de cette thèse comment accéder aux attributs d'une classe (fonction *val?* dans la figure 3.6) afin d'implémenter les attachements procéduraux.

Filtrage

Un deuxième moyen d'inférence est le *filtrage*. Le filtrage consiste à sélectionner les instances qui vérifient un ensemble de conditions appelé *filtre*. Les instances trouvées peuvent être des valeurs d'attributs indéterminés. Le filtre est également représenté par un schéma de classe de `Shirka` (voir fig. 3.7).

La figure 3.7 montre un exemple de filtre, situé dans la facette *$sib-filtre*, qui permet de calculer la valeur de l'attribut *enfants* de la classe **Personne**. Les enfants d'une personne sont les personnes qui ont pour père la personne dont il est question. L'attribut *lui-même*

[5] La liste complète de ces facettes est donnée par le manuel d'utilisation de `Shirka` [Rechenmann et al., 1990].

[6] Toute procédure est une sous-classe de la classe générique **Méthode** de `Shirka`.

```
{ Personne
    sorte-de        =             Objet      ;
    lui-même        $var-nom      lui        ;
    a-pour-père     $un           Personne   ;
    enfants         $liste-de     Personne
                    $sib-filtre
                          {  Personne
                                lui-même      $var ->   enfants ;
                                a-pour-père   $var <-   lui       }
}
```

Figure 3.7 - : Exemple d'un filtre appartenant à la classe **Personne**. Ce filtre détermine la valeur de l'attribut *enfants* de cette classe.

est implicitement défini dans tout schéma de classe. Pour une instance de cette classe, sa valeur est l'instance elle-même.

Classification d'instances

Le dernier moyen d'inférence est la *classification d'instances*. Il consiste à déterminer la classe d'appartenance d'une instance. Le principe est de faire descendre une instance dans une classe plus spécifique que celle à laquelle elle est initialement liée, puis à l'y rattacher. Pour ce faire, les valeurs inconnues sont inférées (en utilisant un des moyens d'obtention de valeurs, l'attachement procédural ou le filtrage) ou demandées à l'utilisateur. Elles sont ensuite testées par rapport aux contraintes associées aux attributs dans les sous-classes examinées. Le résultat du processus de la classification est constitué par la liste des classes auxquelles l'instance peut être rattachée (appelées classes sûres), la liste des classes auxquelles l'instance ne peut pas être rattachée (appelées classes impossibles) et la liste des classes auxquelles elle pourrait être rattachée (appelées classes possibles), certaines valeurs d'attributs n'étant pas disponibles pour répondre de façon certaine.

Supposons que l'on ait le graphe de spécialisation illustré par la figure 3.5, et que l'on souhaite classer l'instance *P0* (fig. 3.3), créée dans la classe **Personne**, et possédant 20 comme valeur d'attribut *âge*. Les deux sous-classes **Salarié** et **Etudiant** sont deux classes possibles. Si l'utilisateur fournit un numéro de carte d'étudiant, *P0* aura comme classe sûre la classe **Etudiant**. La classe **Salarié** reste incertaine car les valeurs de ses deux attributs *salaire* et *nom-établissement* ne sont pas disponibles.

Cette fonctionnalité de classification est la plus importante d'une base de connaissances à objets [Rechenmann, 1988]. De plus, elle ne possède pas d'équivalent dans les bases de données[7].

Nous venons de terminer la description du formalisme utilisé pour représenter les connaissances dans les bases. Nous allons désormais nous intéresser à leurs versions.

[7] La mutation des objets d'une classe à une autre est une opération différente de la classification. D'une part, c'est une opération imposée par l'utilisateur. D'autre part, cette opération n'est pas toujours possible sans effectuer des modifications sur l'objet concerné (par exemple le convertir conformément à sa nouvelle classe, rompre ses liens avec les autres objets, etc.).

3.3 Modèle choisi pour la conception des versions

Dans cette section, nous décrivons le modèle proposé pour représenter les versions des bases de connaissances et pour les gérer. Dans la conception de notre architecture, nous avons cherché à répondre aux besoins suivants :

1. Respecter la généralité du modèle de versions et son indépendance totale par rapport au modèle de représentation de connaissances. Cette propriété d'orthogonalité est présente dans d'autres travaux [Reichenberger, 1989] [Cellary et Jomier, 1994].

2. Prendre en compte la taille importante d'une base de connaissances. Une BC, et en particulier la base consensuelle (BCons), est censée contenir toutes les connaissances possibles sur un domaine particulier. On se place donc résolument dans le contexte des grandes bases de connaissances. Ce point très délicat doit être pris en considération dans l'architecture des versions d'autant que ces dernières représentent des bases entières à un moment donné. Mais les versions ne doivent pas être de taille aussi importante que les bases correspondantes.

3. Permettre la construction incrémentale d'une base de connaissances. La base est complétée par étapes en ajoutant de nouvelles connaissances et en modifiant des connaissances déjà existantes. L'opération de modification doit être vue comme une sorte d'augmentation et non pas une destruction de l'ancien contenu. Nous devons donc prévoir une architecture adaptée à ce mode de construction.

Une version de la base de connaissances est un empilement de couches. Chaque couche représente la différence entre deux versions successives. Nous décrivons tout d'abord les couches en précisant quand et pourquoi elles sont créées et en détaillant leur contenu. Nous définissons ensuite la relation qui existe entre les couches et les versions permettant d'élaborer ces dernières à partir des premières.

3.3.1 Description de la notion de couche

La *couche* est le dispositif de stockage permettant d'établir les versions de bases de connaissances. Elle constitue donc un moyen physique d'implémentation des versions, la version étant la seule entité visible pour l'utilisateur.

L'idée des couches est inspirée de `PIE` [Goldstein et Bobrow, 1980]. `PIE` est un environnement pour le développement interactif de programmes. La caractéristique importante de `PIE`, qui est concrétisée par la notion de couches, est l'incrémentalité et le versionnement. Ces aspects ont justifié notre choix.

Dans notre système, les couches sont empilées les unes au dessus des autres, de façon à ce que toute entité dans une couche masque la même entité, si elle existe, dans sa couche inférieure (en notant que la première couche créée est une couche inférieure de la deuxième, etc.)[Tayar, 1993]. Elles sont gérées et créées automatiquement et uniquement par le système de façon complètement transparente vis-à-vis de l'utilisateur (à la différence de `PIE` qui autorise l'utilisateur à créer ses propres couches).

Le processus de création de couches effectué par le système est le suivant :

- lors d'une demande de création d'une base de connaissances par l'utilisateur, le système de versions crée la première couche correspondante. Il insère, dans celle-ci, les schémas des classes et des instances fournis par l'utilisateur et décrivant ses connaissances;

- lorsque la base doit être mise à jour, l'apport des modifications[8] concernant le contenu de la dernière couche nécessite la création automatique d'une nouvelle couche. Cette dernière devient alors la couche courante, c'est-à-dire celle qui recevra les nouvelles connaissances acquises, modifiées ou supprimées. Ces mises à jour dans la base peuvent perturber le contenu des couches précédentes. Le système redéfinit alors dans la couche courante toutes les conséquences de ces modifications(voir fig. 3.8). Ces conséquences peuvent toucher à la fois la description d'une classe et les valeurs des attributs d'une instance. La section 3.5 est consacrée entièrement à la gestion de la cohérence d'une couche et plus généralement celle d'une version.

Lorsqu'un utilisateur achève son travail avec sa base, il sauve les modifications effectuées. La couche courante est alors fermée par le système. Elle est considérée figée et ne peut plus être modifiée. Elle constitue avec les couches antérieures un historique de la construction de la base.

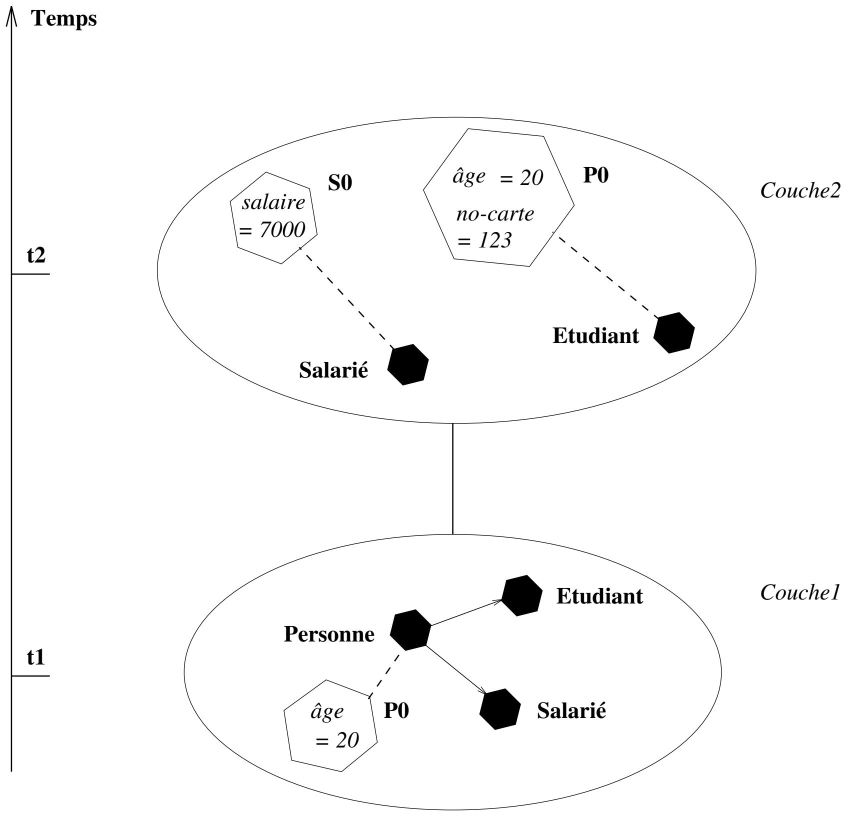


Figure 3.8 - : Description du contenu des couches. La couche supérieure *Couche2* contient les connaissances qui la différencient de sa couche inférieure *Couche1*, ainsi que les connaissances nécessaires pour le maintien de sa cohérence.

La figure 3.8 montre un exemple de contenu de couches. *Couche1* représente la première couche créée à l'instant t_1. Elle contient un graphe de spécialisation (les flèches représentent des relations de spécialisation) dont la racine est la classe **Personne**. Une

[8]Les modifications comportent le changement, l'addition ou la suppression d'une instance ou d'une classe de la base. Dans la section 3.5, nous exposons toutes les opérations qu'un utilisateur peut faire sur une base.

instance de celle-ci, *P0*, a aussi été créée dans *Couche1* (les lignes pointillées représentent le lien d'appartenance d'une instance à une classe). La deuxième couche, *Couche2*, a été créée au-dessus de *Couche1* (i.e. *Couche1* est la couche inférieure de *Couche2*). On y trouve les changements apportés à la base à l'instant t_2 : l'ajout de l'instance *S0* et la modification de *P0*. Cette couche doit également incorporer les conséquences de la mise à jour de la base. C'est pourquoi, la classe **Salarié** qui possède une instance supplémentaire a été ré-insérée, de même que la classe **Etudiant** dont la liste d'instances a été modifiée. La classe **Personne** n'a pas à être redéfinie dans *Couche2* car elle n'a pas été modifiée puisqu'elle possède toujours l'instance *P0* (selon le principe de *l'inclusion des instances*, voir §3.2).

3.3.2 Définition formelle d'une couche

Une couche peut être modélisée par l'union de sa liste des schémas supprimés et de sa liste des schémas ajoutés :

$$couche[i] = \{\text{l-schémas-supprimés}\}[i] \cup \{\text{l-schémas-ajoutés}\}[i] \tag{3.1}$$

Les trois opérateurs élémentaires sur les couches correspondent alors au traitement suivant :

- *ajout d'une entité (classe ou instance)* : on ajoute la nouvelle entité à la liste {l-schémas-ajoutés};
- *suppression d'une entité (classe ou instance)* : on ajoute l'entité supprimée à la liste {l-schémas-supprimés};
- *modification d'une entité (classe ou instance)*[9] : on ajoute l'entité initiale à la liste {l-schémas-supprimés} et la nouvelle entité (après modification) à la liste {l-schémas-ajoutés}. L'instance *P0*, figure 3.8, va par exemple faire partie des deux listes de sa couche. Elle va être insérée dans {l-schémas-supprimés}[2] et dans {l-schémas-ajoutés}[2].

Nous pouvons exprimer la différence (ensembliste) entre deux versions successives. La différence entre v[i] et v[i-1] est par définition l'ensemble des schémas qui appartiennent à v[i] et qui n'appartiennent pas à v[i-1]. Il s'agit donc des schémas qui ont été ajoutés au niveau de v[i] ainsi que ceux qui y ont été modifiés. Ceci correspond à {l-schémas-ajoutés} de couche[i].

$$v[i] \setminus v[i-1] = \{\text{l-schémas-ajoutés}\}[i] \tag{3.2}$$

De même, la différence entre v[i-1] et v[i] est l'ensemble des schémas qui sont dans v[i-1] et non pas dans v[i]. Ce sont donc les schémas qui ont été supprimés ou modifiés au niveau de v[i]. Cette différence correspond à la liste {l-schémas-supprimés} de couche[i].

$$v[i-1] \setminus v[i] = \{\text{l-schémas-supprimés}\}[i] \tag{3.3}$$

D'après les équations 3.1, 3.2 et 3.3 on obtient :

$$couche[i] = v[i] \setminus v[i-1] \cup v[i-1] \setminus v[i] = v[i] \Delta v[i-1] \tag{3.4}$$

L'équation 3.4 montre qu'une couche correspond à la différence symétrique[10] entre deux versions successives. Cette relation permet au système de les comparer facilement.

[9]Au niveau de l'implémentation, la modification n'est pas traitée comme une suppression et un ajout pour des raisons de mémoire.

[10]Opération ensembliste.

3.3.3 Relation entre versions et couches

Comme nous l'avons déjà vu, chaque couche ne mémorise que les informations qui la différencient de sa couche inférieure. Pour travailler sur l'état initial de la base, le système extrait donc le contenu de la première couche. Pour consulter l'état le plus récent, la dernière couche ne suffit pas car elle ne donne qu'une vue intermédiaire de la base. Le système est alors obligé de prendre en compte toutes les couches. Ceci est un point de divergence avec `PIE` puisqu'une couche, dans ce dernier environnement, contient la totalité des informations permettant de l'utiliser seule indépendamment des autres couches. Un état[11] d'un programme est défini comme n'importe quelle séquence de couches. Comme il n'existe pas de restriction sur l'ordre au sein d'une séquence, il est possible d'en engendrer une ne possédant pas de sens. Dans le cadre de notre système, cette liberté de combinaison de couches est rigoureusement interdite puisque les états de la base reflètent l'historique de son développement. Le $n^{\text{ème}}$ état de la base ne correspond pas uniquement à n couches mais aux n premières couches possédant un ordre précis.

Nous devons dans notre cas mettre en place un mécanisme transparent à l'utilisateur permettant, lorsque celui-ci accède à une version, de récupérer les informations nécessaires présentes dans les couches inférieures (et qui ne sont pas présentes dans la couche correspondante). Nous allons déduire une relation qui permette de reconstituer une version à partir des couches.

$\forall$ A et B deux ensembles, nous avons les relations ensemblistes générales :

$$A = (A \setminus B) \cup (A \cap B) \tag{3.5}$$

$$A \cap B = A \setminus (A \setminus [A \cap B]) \tag{3.6}$$

$$A \setminus (A \cap B) = A \setminus B \tag{3.7}$$

Selon l'équation 3.5, une version v[i] peut être donnée par :

$$v[i] = (v[i] \setminus v[i-1]) \cup (v[i] \cap v[i-1]) \tag{3.8}$$

Selon l'équation 3.6, nous avons :

$$v[i-1] \cap v[i] = v[i-1] \setminus (v[i-1] \setminus (v[i-1] \cap v[i])) \tag{3.9}$$

D'après l'équation 3.7, nous avons :

$$v[i-1] \setminus (v[i-1] \cap v[i]) = v[i-1] \setminus v[i] = \{\text{l-schémas-supprimés}\}[i] \tag{3.10}$$

La liste {l-schémas-supprimés} peut être réécrite en :

$$\{\text{l-schémas-supprimés}\}[i] = v[i-1] \cap couche[i] \tag{3.11}$$

D'après cette équation, l'équation 3.10 est remplacée par :

$$v[i-1] \setminus (v[i] \cap v[i-1]) = v[i-1] \cap couche[i] \tag{3.12}$$

Tenant compte de cette équation, l'équation 3.9 est équivalente à :

$$v[i] \cap v[i-1] = v[i-1] \setminus (v[i-1] \cap couche[i]) \tag{3.13}$$

[11] Appelé “context” dans `PIE`.

Selon l'équation 3.7, nous pouvons écrire :

$$v[i-1] \setminus (v[i-1] \cap couche[i]) = v[i-1] \setminus couche[i] \tag{3.14}$$

L'équation 3.13 est remplacée par :

$$v[i] \cap v[i-1] = v[i-1] \setminus couche[i] \tag{3.15}$$

En fonction des équations 3.2 et l'équation 3.15, nous pouvons remplacer l'équation 3.8 par :

$$v[i] = \{\text{l-schémas-ajoutés}\}[i] \cup (v[i-1] \setminus couche[i]) \tag{3.16}$$

Par récursivité, nous pouvons écrire :

$$\begin{aligned} v[i] &= \{\text{l-schémas-ajoutés}\}[i] \cup (v[i-1] \setminus couche[i]) && (3.17)\\ v[i-1] &= \{\text{l-schémas-ajoutés}\}[i-1] \cup (v[i-2] \setminus couche[i-1]) && (3.18)\\ \ldots & && (3.19)\\ v[2] &= \{\text{l-schémas-ajoutés}\}[2] \cup (v[1] \setminus couche[2]) && (3.20)\\ v[1] &= couche[1] && (3.21) \end{aligned}$$

D'après ces relations, nous sommes en mesure de calculer n'importe quelle version (i) à partir de l'ensemble des couches comprises entre couche[1] et couche[i]. Nous appelons ce mécanisme le *quasi-héritage*. Nous reviendrons à ce mécanisme au cours de la section suivante.

L'inconvénient d'une telle approche est la nécessité de réaliser des opérations d'union et de différence entre les couches afin de se déplacer au sein des versions. Afin d'alléger le mécanisme de calcul, la version la plus récente est reconstituée par défaut par le système chaque fois que la base correspondante est chargée. L'utilisateur travaille en effet plus souvent sur cette version que sur les précédentes.

3.3.4 Mécanisme de quasi-héritage

Le *quasi-héritage*, défini entre les couches[12], est analogue à la notion d'héritage dans les langages à objets [Masini et al., 1989], dans le sens où chaque couche partage ou factorise de l'information avec sa couche inférieure. Les connaissances sont donc propagées, si elles ne sont pas redéfinies, de la première couche à la deuxième et plus généralement de la $i^{\text{ème}}$ à la $(i+1)^{\text{ème}}$ (où i est le numéro de la couche indiquant son ordre de création). En outre, chaque couche supérieure peut masquer sa couche inférieure en modifiant des schémas de classes ou des schémas d'instances de sa couche inférieure. La mise en œuvre de cette relation est dynamique et se fait lorsque l'utilisateur demande à accéder à un état de la base (consultation, mise à jour).

Dans la figure 3.9, la base de connaissances possède trois états (versions) différents correspondant aux trois couches. Dans le premier état, la BC est constituée d'un seul graphe de spécialisation dont la racine est la classe **Personne** et les feuilles sont les classes **Salarié** et **Etudiant**. La classe **Personne** possède une instance $P0$. Si on consulte le deuxième état de la BC, nous retrouvons les trois classes (**Personne, Salarié**). La classe **Salarié** possède une instance *S0* et la classe **Personne** possède les deux instances : $P0$ et $S0$

[12] Le quasi-héritage est la relation défini entre les couches, la relation entre les versions reste celle de la dérivation

(selon le principe d'*inclusion des instances* du modèle `Shirka`). Pour l'état le plus récent lié à la dernière couche, le contenu de la BC est le suivant : quatre classes (**Personne, Salarié, Etudiant, Etudiant-diplômé**) et deux instances : $S0$ (appartenant à **Salarié** et **Personne**), $P0$ (appartenant à la classe **Personne**).

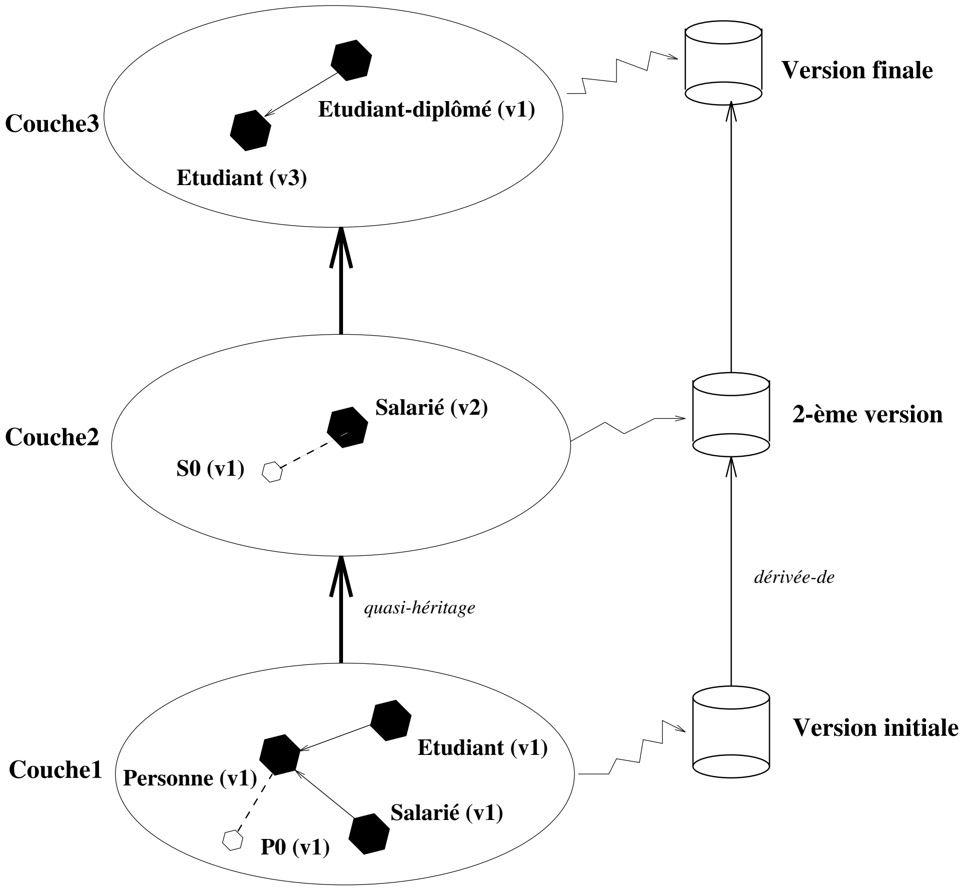


Figure 3.9 - : La relation *quasi-héritage* entre les couches permettant d'associer à chaque couche sa version correspondante de la base de connaissances. La relation entre les différentes versions est celle de *dérivée-de*.

Enfin, dans les langages à objets, lorsqu'une classe **B** hérite d'une autre classe **A**, cela signifie que **B** est une spécialisation de **A**. Une classe en spécialise une autre en ajoutant de nouvelles connaissances ou en raffinant celles de ses sur-classes mais jamais en les remettant en cause. Dans notre cas, selon cette définition, une couche héritant des données de sa couche inférieure peut ne pas être une spécialisation de celle-ci. En effet, les modifications apportées dans une couche peuvent ne pas être seulement un affinement des connaissances de sa couche inférieure. Elles peuvent très bien être une révision du contenu de cette dernière. Par exemple, un schéma (d'une classe ou d'une instance) d'une couche inférieure peut être supprimé dans une de ses couches supérieures. Il peut même être remplacé par un autre schéma ayant le même nom sans avoir la même liste d'attributs.

Jusqu'à présent, nos versions ressemblent à des révisions car la nouvelle version est toujours dérivée de la version courante ce qui engendre une liste. L'ordre défini est donc un ordre total. Notre système n'est toutefois pas limité à cet ordre et est capable de gérer des structures non-linéaires entre les versions de la base. Ceci constitue le sujet de la section suivante.

3.3.5 Description de la notion d'axe de travail

Nous allons présenter, au cours de cette section, un arbre de versions concurrentes avec un ordre partiel.

Lorsqu'il s'agit de gérer des versions concurrentes, les principaux auteurs parlent le plus souvent d'hypothèses [Katz, 1990]. Pour nous, accepter les nouvelles informations soumises à titre d'hypothèse signifie que celles-ci ne peuvent être ni infirmées ni confirmées étant donné le contenu courant de la base. Offrir cette possibilité est une nécessité car dans un mode de construction partagée, différents chercheurs peuvent avoir des résultats expérimentaux différents qui doivent néanmoins pouvoir tous être intégrés dans la base concernée. Des connaissances hypothétiques à un instant donné peuvent s'avérer être des connaissances valides ultérieurement (lorsque la base a suffisamment progressé pour pouvoir les juger).

Il faut signaler qu'il n'y a pas de conflit entre la notion de connaissances hypothétiques et celle de connaissances consensuelles. La majorité de la communauté admet qu'il y a un désaccord portant sur les nouvelles connaissances acquises sans pouvoir trancher en faveur d'une hypothèse ou d'une autre étant donné l'état actuel des connaissances.

Définition

La coexistence de deux hypothèses se caractérise par un embranchement conduisant à deux couches séparées. C'est la solution proposée afin de pouvoir supporter des données contradictoires au sein d'une même base. Par définition, nous appelons *axe de travail* une branche dans le graphe de versions. Ces axes sont les équivalents des variantes dans les autres modèles de la littérature.

La première version de chaque axe est la version racine. La dernière version est une des feuilles (remarquons que le nombre des axes dans un arbre est égal au nombre des feuilles dans celui-ci). Ceci est illustré par la figure 3.10 où il y a 5 couches, 5 versions et deux axes. Nous nous sommes limités, au cours de cette thèse, à des arbres de versions (et non pas des graphes). Ainsi, une version ne peut être dérivée que d'une seule version inférieure. Finalement, chaque axe a un nom unique, donné par l'utilisateur qui est à l'origine de sa création, qui le différencie des autres axes qui sont présents dans la même base.

Conséquences sur la gestion de la base

L'introduction de la notion d'axe (ou hypothèse) de travail a plusieurs conséquences :

1. Pour accéder à une version d'une classe (ou plus généralement à une version de la base), l'utilisateur doit fournir à la fois le nom de l'axe d'appartenance de la version désirée ainsi que le numéro de celle-ci. Ces deux informations définissent ensemble ce qu'on appelle le contexte de travail courant. Si le nom de l'axe n'est pas spécifié, l'axe utilisé par défaut est celui contenant la dernière version créée.

2. À partir du moment où un axe peut être vu comme une liste de versions, le problème revient, une fois que l'axe est déterminé, à travailler avec des versions linéaires (comme nous l'avons indiqué auparavant).

Selon cette description, les couches présentent ici un double intérêt. D'une part, elles sont adaptées à notre mode de construction de BC car elles offrent la possibilité de regrouper les changements incrémentaux. D'autre part, elles nous permettent de supporter les

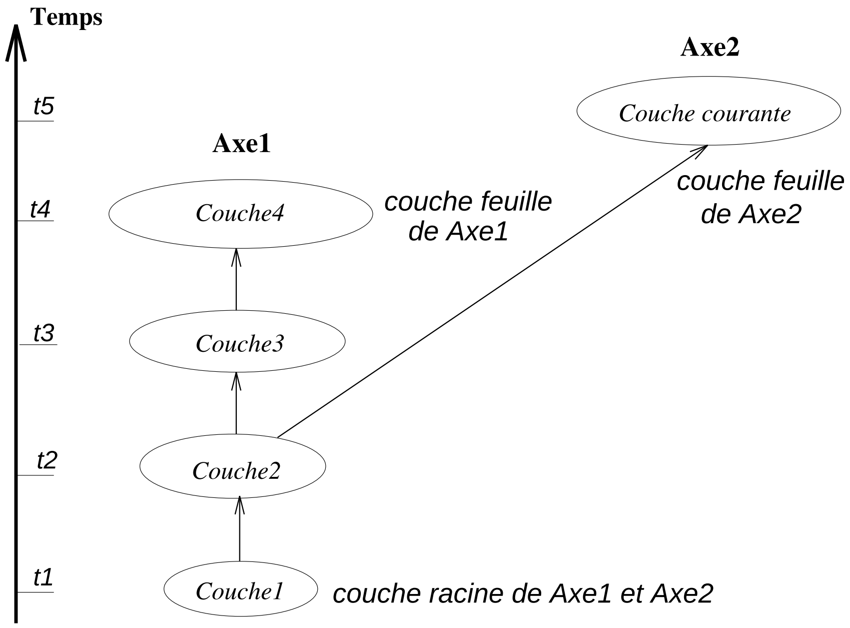


Figure 3.10 - : Un arbre de couches. Chaque branche dans celui-ci partant de la couche racine jusqu'à une couche feuille représente un axe de travail. La couche courante, par défaut, est la dernière couche créée.

versions et les variantes de celles-ci. Dans ce qui suit, nous expliquons comment s'effectue la gestion des opérations de manipulation.

3.4 Opérations de manipulation de versions

La manipulation d'une version équivaut à la manipulation d'un état de la base à un moment donné. Pour cette raison, le mot "base" ou "version" désigne, dans ce qui suit, la même entité. Nous allons nous intéresser aux deux types de manipulation : la modification au travers de la création et de l'ajout des versions et la consultation au travers de l'identification et de la sélection d'une version. Un utilisateur peut mettre à jour ou interroger les versions de sa BC si et seulement si cette dernière est déjà chargée dans son espace de travail.

3.4.1 Opérations de création et d'ajout

Dans cette section, nous allons décrire les différentes opérations de création possibles : la création d'une base, d'un axe de travail ou d'une version. Mais tout d'abord, il est important de signaler la manière adoptée par le système pour numéroter une nouvelle version créée au sein de la même base. Ce numéro va être utiliser comme identificateur possible d'une version.

Numérotation des versions

Si nous regardons du côté de la littérature, la plupart des gestionnaires utilisent le numéro de la version comme identificateur (physique et logique). Ce numéro est généré automatiquement lors de la création d'une nouvelle version. Ils représentent, dans les travaux de [Chou et Kim, 1986] et [Dittrich et Lorie, 1988], l'ordre de sa création. Par exemple, un utilisateur désirant choisir une version plus récente que la version numéro 2, doit fournir un numéro plus élevé. L'utilisation de cette façon de numéroter ne permet pas par contre de savoir si la version 2 est une variante de 3 ou de 4.

L'extension d'une telle numérotation a pour but de permettre d'attribuer aux versions des sous-séquences de numéro (i.e. 1.2, 2.3.1). La $i^{\text{ème}}$ version, dérivée d'une version dont le numéro est N se voit affecter le numéro $N.i$ (voir SCCS [Rochkind, 1975]). Par exemple, la version 3.2.1 est la première version dérivée de la version 3.2 et les versions 1.1 et 1.2 sont deux variantes de la version 1. L'avantage d'un tel ordre lexicographique[13] est que les numéros sont significatifs. Par contre, la numérotation risque d'être difficile à maintenir lorsque le graphe de versions se développe en profondeur plus rapidement qu'en largeur. Les numéros peuvent être assez longs (par exemple 1.2.1.3.1.2) et deviennent illisibles. Pour cette raison, cette numérotation ne convient pas dans le cadre d'une problématique d'élaboration d'une base de connaissances consensuelles. En effet les points de vue divergents (les axes de travail ou la largeur du graphe) seront peu nombreux par rapport aux versions développées incrémentalement représentant la hauteur du graphe.

Par opposition à une numérotation globale à la base, nous avons choisi une numérotation locale à chaque axe de travail. Ce numéro attribué à chaque version correspond à l'ordre de création de la version au sein de son axe d'appartenance (une version peut appartenir à plusieurs axes mais son ordre de création reste invariant). Cette manière de numéroter permet de savoir, pour la même branche, quelle version est dérivée d'une autre (par exemple, version 2 est créée avant version 3). D'autre part, la relation de concurrence entre les versions est facile à identifier. Il suffit de remarquer que deux versions possédant le même numéro appartiennent à deux axes différents (i.e. elles sont deux variantes).

Création d'une base et de ses versions

Lors de la demande de création d'une BC, le système de versions initialise l'arbre des versions correspondant avec une seule branche (axe), dont le nom est demandé à l'utilisateur, et un seul nœud (la première couche) auquel sont associés sa date de création ainsi que le numéro 1. Le contenu de cette couche sera la version initiale des connaissances (les schémas de classes et d'instances) de l'utilisateur.

Nous allons exposer le processus de création à travers un exemple. Nous étudions le cas d'une base ayant deux axes de travail $A1$ et $A2$. L'axe $A1$ a trois versions $V1$, $V2$ et $V3$ tandis que $A2$ n'en a que deux $V1$ et $V2$ (fig. 3.11). Supposons que $V2$ dans $A2$ soit la dernière version créée. Après avoir chargé la base, l'utilisateur souhaite la modifier. Il doit alors choisir la version qu'il souhaite modifier. Selon la version choisie, le système est face à trois possibilités :

1. l'utilisateur n'identifie pas une version particulière. Dans ce cas, une nouvelle version est dérivée de la version courante par défaut (i.e. $V2$ de $A2$). Son numéro sera 3 et elle sera la version courante (cas1 figure 3.12);

2. l'utilisateur choisit de modifier $V3$ de $A1$. Comme il s'agit d'une version feuille, la nouvelle version créée ($V4$) va être ajoutée à l'axe $A1$ et aura 4 comme numéro. Elle constitue avec $A1$ le contexte de travail courant (cas2 figure 3.12);

3. l'utilisateur identifie une version antérieure (par exemple $V2$ de $A1$). Ceci signifie que l'utilisateur a besoin d'avoir une autre version dérivée de $V2$ mais pas dans le

[13] Cette numérotation est appelée numérotation de Deweg.

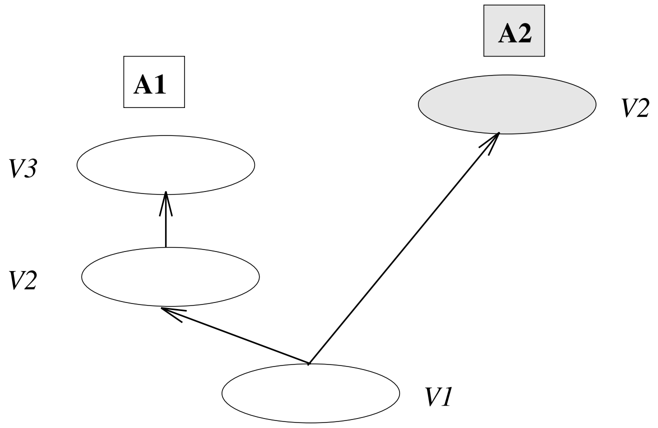


Figure 3.11 - : Exemple d'arbre de versions avec comme version courante *V2* et axe courant *A2*.

même axe qu'elle (i.e. $A1$). Cette nouvelle version ($V3$) va donc être un descendant de $V2$ mais dans un autre axe de travail. Dans ce cas, l'utilisateur doit donner le nom de celui-ci ($A3$ par exemple). $V3$ sera la nouvelle version courante et $A3$ l'axe courant. Un axe est créé chaque fois que la version modifiée n'est pas une version feuille dans l'arbre de versions (cas3 figure 3.12).

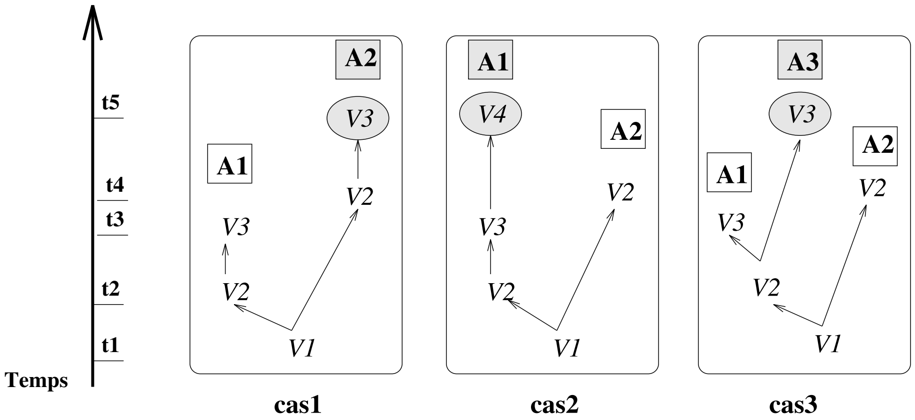


Figure 3.12 - : Les trois possibilités de modification de l'arbre de versions de la figure précédente.

Remarque : nous supposons que la date de création d'un axe de travail est celle de la version qui est à l'origine de sa création. La date de création de l'axe $A1$ est celle de $V1$. Pour $A2$ et $A3$, il s'agit de la date de création de leurs deux versions respectives $V2$ et $V3$.

3.4.2 Opérations d'identification et de sélection

Identification et sélection d'une version de la base

L'utilisateur peut identifier de manière unique une version de la base en fournissant une des informations suivantes :

- la date de création de la version. Cette date est un identificateur explicite car elle est exprimée dans chaque version créée (voir structure de la classe version 5.1.2). Cet identificateur est global.

- le nom de l'axe de travail et le numéro de la version dans cet axe. Ce numéro n'est pas stocké implicitement dans une version, mais il correspond à son ordre de création dans l'axe déterminé. Cet identificateur est local (à l'axe de travail).

- le nom de la version. Ce nom est donné par l'utilisateur lors de la création d'une version. Il doit être unique au sein de la même base.

L'opération de sélection a pour but de déterminer quelle version sera la version courante. L'utilisateur peut sélectionner une version (en donnant une des informations ci-dessus). Il peut également choisir relativement à la version courante actuelle une version appartenant au même axe de travail. Il fournit pour cela une des informations suivantes :

- son numéro relatif par rapport à la version courante en utilisant les deux caractères *"-"* et *"+"* respectivement pour les versions antérieures et pour les versions postérieures. Si la version courante a le numéro 6 alors -2 désigne la version numéro 4 et $+$ rend la septième, etc.

- en utilisant les deux mots-clés, **Precedente** et **Successeur**, fournis par le système pour chacune des versions qui respectivement précède et suit la version courante.

En outre, l'utilisateur a la possibilité de définir des critères de sélection sur les versions de la base. Le système choisira l'ensemble de versions répondant à ces critères. En fait, ces derniers ne sont que des requêtes posées sur les attributs de la classe **Version**. Les détails sont donnés dans la section 5.3.2.

Remarque : l'utilisateur peut identifier une version en précisant un temps t. S'il n'existe pas une version créée à ce temps, la réponse du système, dans ce cas, est la première version créée à un temps supérieur à t. Si t précède la date de création de la première version, alors cette dernière est choisie.

Identification et sélection d'une version d'une entité

Après avoir choisi une version, l'utilisateur peut extraire les connaissances voulues grâce aux noms de la classe et de l'instance (formalisme `Shirka`). Il accède alors à la version de ces connaissances correspondant à la version sélectionnée de la base.

Les trois expressions suivantes sont des exemples de requêtes :

A1.*.Personne.l-sous-classe
A1.2.P0.âge
t.Personne.l-inst

La première permet de trouver la liste des sous-classes de la classe **Personne** par rapport à la version courante de l'axe $A1$. Notons que dans ce cas, le numéro de la version n'a pas à être précisé. La deuxième interrogation consiste à trouver la valeur de l'attribut *âge* de la deuxième version de l'instance $P0$ dans l'axe $A1$. Cette valeur sera calculée par un attachement procédural (ou un filtre) si nécessaire et la réponse sera envoyée au demandeur. Par contre, cette valeur ne sera pas stockée et la base ne changera pas d'état (car il s'agit d'une opération de consultation). Si l'utilisateur désire garder la valeur inférée, il doit alors demander d'effectuer une mise à jour de la base (Cf. §3.5). La dernière requête rend la liste des instances de la classe **Personne** au moment donné t (dans ce cas, le nom d'un axe n'a pas à être précisé).

Une instance peut être sélectionnée en spécifiant une condition particulière sur un (ou plusieurs) de ses attributs. Afin de sélectionner la *personne* dont l'*âge* est égale à 20 ans dans la version de la base qui *précède* la version courante de l'axe *A1*, l'utilisateur doit taper la requête :

A1.precedente.[X.(âge = 20) et X.(est-un = Personne)]

La sélection d'une instance peut mettre en jeu plusieurs versions de la base. Plus précisément, l'utilisateur peut demander de trouver toutes les personnes dont l'âge est égal à 20 (en posant la même requête que ci-dessus sans identifier une version particulière). Dans ce cas, toutes les versions de la base (appartenant à l'axe choisi) sont prises en considération. Elles doivent être balayées afin de filtrer l'ensemble des instances voulues. Pour ce type de requêtes, nous avons pris la décision d'ajouter un paramètre temporel : l'utilisateur doit préciser à partir de quel moment il veut accéder à l'information demandée afin de rendre l'algorithme de recherche plus rapide. Le système offre plusieurs opérateurs :

- ***depuis*** <temps>
- ***entre*** <temps1> <temps2>
- ***jusqu'à*** <temps>

Pour avoir l'ensemble des salaires touchés par l'instance *P0* de la classe **Personne** (dans l'axe *A*1) jusqu'en 1995, l'utilisateur doit par conséquent taper la requête :

jusqu'à 1995 {**A1.[P0.salaire et P0.(est-un = Personne)]**}

Le langage de requêtes est exposé au cours de la section 5.3 où d'autres opérateurs temporels sont fournis. L'exécution de ces requêtes est donnée § 5.3.3.

Après avoir exposé la mise en œuvre des opérations de manipulation des versions de la base, nous détaillons dans ce qui suit les aspects liés à la gestion et au maintien de la cohérence d'une version.

3.5 Gestion de la cohérence d'une version

Comme nous l'avons déjà montré, une couche contient les nouvelles connaissances apportées à la base ainsi que les schémas des couches précédentes modifiés par l'intégration de ces connaissances. Au cours de cette section, nous allons (1) examiner toutes les opérations de changement qu'un utilisateur peut réaliser, (2) expliquer les conséquences sur la base et enfin (3) décrire les mesures (au niveau couche) que doit prendre le système de versions. Tout ce travail définit la gestion de la cohérence. Cette gestion dépend fortement du modèle choisi pour la représentation des connaissances car c'est ce dernier qui détermine le comportement de la base vis-à-vis d'un changement effectué. Nous supposons que ***C*** est la couche courante et d'après l'étude du modèle `Shirka`, nous avons pu distinguer trois groupes d'opérations : les opérations portant sur les instances(§ 3.5.1), les opérations de modification des références entre les instances(§ 3.5.2) et finalement les opérations portant sur les classes(§ 3.5.3). L'annexe A contient les traces d'exécution de ces trois groupes d'opérations.

3.5.1 Opérations sur les instances

La création d'une instance : elle implique l'insertion de cette instance dans ***C***. La création de cette instance provoque le changement de la liste des instances de sa classe d'appartenance. Cette dernière doit donc être également insérée dans C (c'est-à-dire qu'une nouvelle version de la classe est créée au niveau de la couche courante).

Si la nouvelle instance a un attribut dont le type est une référence à un schéma de classe (le cas de l'instance $P0$, dans la figure 3.13, dont les trois attributs *enfants*, *date-naissance* et *adresse* prennent leur valeur respectivement dans les classes **Personne**, **Date** et **Adresse**), il existe trois possibilités :

```
{Personne
    sorte-de          =             Objet     ;
    nom               $un           chaine    ;
    enfants           $liste-de     Personne  ;
    date-naissance    $un           Date      ;
    adresse           $un           Adresse   }

{Date
    sorte-de          =             Objet     ;
    jour              $un           entier    ;
    mois              $un           chaine    ;
    année             $un           entier    }

{Adresse
    sorte-de          =             Objet     ;
    numéro            $un           entier    ;
    rue               $un           chaine    ;
    ville             $un           entier    }
```

```
{ P0
    est-un            =    Personne       ;
    date-naissance    =
       { est-un       =    Date           ;
         jour         =    15             ;
         mois         =    "septembre"  } ;
    adresse           =    A0
}

{A0
         est-un       =    Adresse
         numéro       =    15             ;
         rue          =    "joseph brun"  }
```

Figure 3.13 - : Exemple d'une base avec trois classes : **Personne**, **Date** et **Adresse** et deux instances $P0$ et $A0$. L'instance $P0$ référence l'instance $A0$ via l'attribut *adresse*. L'attribut *enfants* de *P0* n'a pas de valeur. L'attribut *date-naissance* a comme valeur une instance de la classe **Date** non nommée. Cette instance existe tant que l'instance *P0* qui la référence existe.

1. l'utilisateur ne fournit de valeur à cet attribut. Aucune mesure est prise au niveau de C (le cas de l'attribut *enfants* de $P0$);
2. l'utilisateur demande que cet attribut soit une référence à une instance identifiée par son nom (le cas de l'attribut *adresse* dont la valeur est l'instance $A0$). Cette instance peut soit ne pas encore être créée, soit déjà exister. Dans les deux cas, il s'agit d'une opération d'attribution de valeur et C n'est pas modifiée;
3. l'utilisateur décide que la valeur de l'attribut est directement une instance; celle-ci va être dynamiquement créée afin de la rattacher à l'attribut (c'est le cas de l'attribut *date-naissance* de $P0$ qui a comme valeur l'instance elle-même et non pas une référence). Cette instance sera non nommée et donc non partageable. Par contre, `Shirka` ajoute cette instance à la liste des instances de sa classe d'appartenance. Cette dernière est alors recopiée au niveau de la couche C. Dans l'exemple de la figure 3.13, il s'agit de la classe **Date**.

La suppression d'une instance : elle a comme conséquence l'insertion de sa classe dans ***C***. Il est indispensable de signaler que cette suppression est logique et non pas physique. Lorsqu'un utilisateur consulte la version courante, il ne retrouve pas l'instance concernée. Pourtant cette dernière demeure pour les versions précédentes. Cette suppression est gérée par l'intermédiaire de la liste *l-schémas-supprimés* définie pour chaque couche créée (Cf. §3.3.2). Le système va ainsi ajouter le nom de l'instance supprimée à la liste de ***C***.

Lorsqu'une instance est supprimée, ses références aux autres instances le sont aussi. En revanche, les instances référencées existent toujours. Si $P0$ dans la figure 3.13 est effacée, alors la classe **Personne** perdra une instance, mais la classe **Adresse** possède toujours son instance $A0$. L'instance de la classe **Date** par contre va disparaître avec $P0$ puisqu'elle n'a pas de nom lui permettant d'être identifiée ou d'être partagée. Elle ne peut donc pas persister dans la base. C'est pourquoi sa classe **Date** sera insérée dans ***C*** car sa liste d'instances vient d'être changée.

Remarque : comme nous l'avons déjà précisé, lorsque la liste des instances d'une classe est modifiée (en ajoutant ou supprimant une instance), seule la classe en question va avoir une nouvelle version. Néanmoins, toutes les sur-classes de celle-ci sont concernées par cette modification (selon le principe d'*inclusion des instances*). Il n'est pas nécessaire de recopier physiquement ces sur-classes dans la nouvelle version de la base. Lors de l'accès à cette version, le mécanisme de *quasi-héritage* (défini entre les couches) créera des versions logiques de ces sur-classes qui, grâce au mécanisme de *l'héritage* (entre les classes), posséderont automatiquement les listes des instances de leurs sous-classes.

L'ajout ou la modification de valeurs dans un attribut d'une instance : c'est un cas particulier de la création d'une instance. Cette opération implique la redéfinition de cette instance au niveau de ***C***, si le changement a été accepté par `Shirka`. Si la valeur ajoutée (ou la nouvelle valeur) est directement une instance (comme dans le cas de l'attribut *date-naissance* de l'instance $P0$ dans la figure 3.13) alors celle-ci, ainsi que sa classe, vont être insérées en même temps dans ***C***.

La suppression de valeurs dans un attribut d'une instance : c'est un cas particulier de la suppression d'une instance. L'instance modifiée doit être redéfinie dans ***C*** sans la valeur supprimée de l'attribut concerné. Si la valeur supprimée est une référence à une instance, alors c'est la référence qui est rompue. Dans le cas où la valeur supprimée est directement une instance alors cette dernière va aussi être effacée puisque sa durée de vie est celle de la valeur qu'elle représente. Sa classe correspondante doit donc être réintégrée dans la couche ***C***.

L'exécution d'une méthode d'attachement procédural : elle a pour but de compléter l'instance par la valeur de l'attribut résultat de cette méthode. Les mesures prises sont identiques à celle de l'ajout d'une valeur dans un attribut.
Le déclenchement d'un filtre est traité de la même façon.

La modification du corps d'une méthode d'attachement procédural : elle a pour conséquence le changement de la façon dont cette méthode a été implémentée. La valeur inférée par la nouvelle implémentation peut être différente de l'ancienne valeur (avant la modification). Cette opération est traitée de manière similaire à la modification de valeur dans un attribut.

La classification d'une instance : elle permet de classer une instance suivant les principes expliqués dans le paragraphe §3.2.2. Les valeurs inconnues des attributs de l'instance sont inférées ou demandées à l'utilisateur. Le système crée une instance temporaire, copie de l'instance à classer, qui reçoit les valeurs manquantes. Le processus de classification est appliqué sur elle. Ce processus examine les sous-classes de sa classe d'origine afin d'identifier celles pouvant l'accepter. Le résultat de la classification est une liste de noms de classes (sûres, possibles et impossibles). L'opération d'affectation éventuelle de l'instance initiale n'est pas réalisée à ce niveau là. Ceci signifie que le système n'insère dans $\boldsymbol{C}$ que l'instance temporaire.

Le rattachement d'une instance : comme nous venons de le voir, la commande de classification ne rattache pas l'instance à classer. C'est à l'utilisateur de le demander explicitement en précisant le nom de la nouvelle classe de l'instance. Avant d'effectuer le rattachement, `Shirka` vérifie si la classe donnée fait partie des classes sûres ou possibles fournies par la classification. Si tel est le cas, l'instance considérée prend les valeurs de l'instance temporaire (voir classification) qui est ensuite supprimée. Deux schémas ont alors été changés : celui de l'instance qui a été classée et celui de la classe de rattachement. Ces deux schémas sont incorporés dans $\boldsymbol{C}$. L'instance considérée reste toujours une instance de sa classe d'origine puisque sa nouvelle classe est par principe une sous-classe de celle-ci (selon la caractéristique de *l'inclusion des instances* (voir §3.2)). C'est pourquoi, la classe d'origine ne va pas être ré-insérée dans $\boldsymbol{C}$.

Après avoir exposé les opérations sur les instances et leurs conséquences sur la couche courante, nous abordons dans la section suivante les opérations sur les instances complexes et le problème de référence.

3.5.2 Instances complexes et le problème des références

Les instances modélisées dans `Shirka` ne sont pas uniquement des instances simples. Elles peuvent être des instances complexes référençant d'autres instances. Nous nous retrouvons face au problème de configuration (Cf. § 2.6). Il s'agit dans notre cas du problème suivant : une instance peut avoir une référence (à travers un attribut) à une autre instance dans une des couches. Par exemple dans la couche[i], l'instance $P0$ référence l'instance $A0$. Lorsque l'instance référencée ($A0$) est modifiée dans une couche[j], *P0* doit référencer, au sein de la couche[j], la nouvelle version de *A0*. Une nouvelle version de $P0$ doit alors être créée dans la couche[j].

Dans notre cas, il n'est pas nécessaire de créer physiquement l'instance $P0$ au sein de la couche[j] (et dans ce cas surcharger cette dernière), la notion de *quasi-héritage* permettant de fournir automatiquement la version[j] de $P0$ (et d'assurer par conséquent la cohérence de la base). Une version de la base, élaborée par le mécanisme de *quasi-héritage*, contient alors l'ensemble de toutes les versions des schémas (une version par schéma) présentes dans la base à un moment donné (voir fig. 3.14). Ceci n'est que la signification du mot contexte proprement dit et qui a été introduit comme solution du problème de référence (Cf. §2.6.4). Les versions d'une base ne sont que des contextes créées dynamiquement.

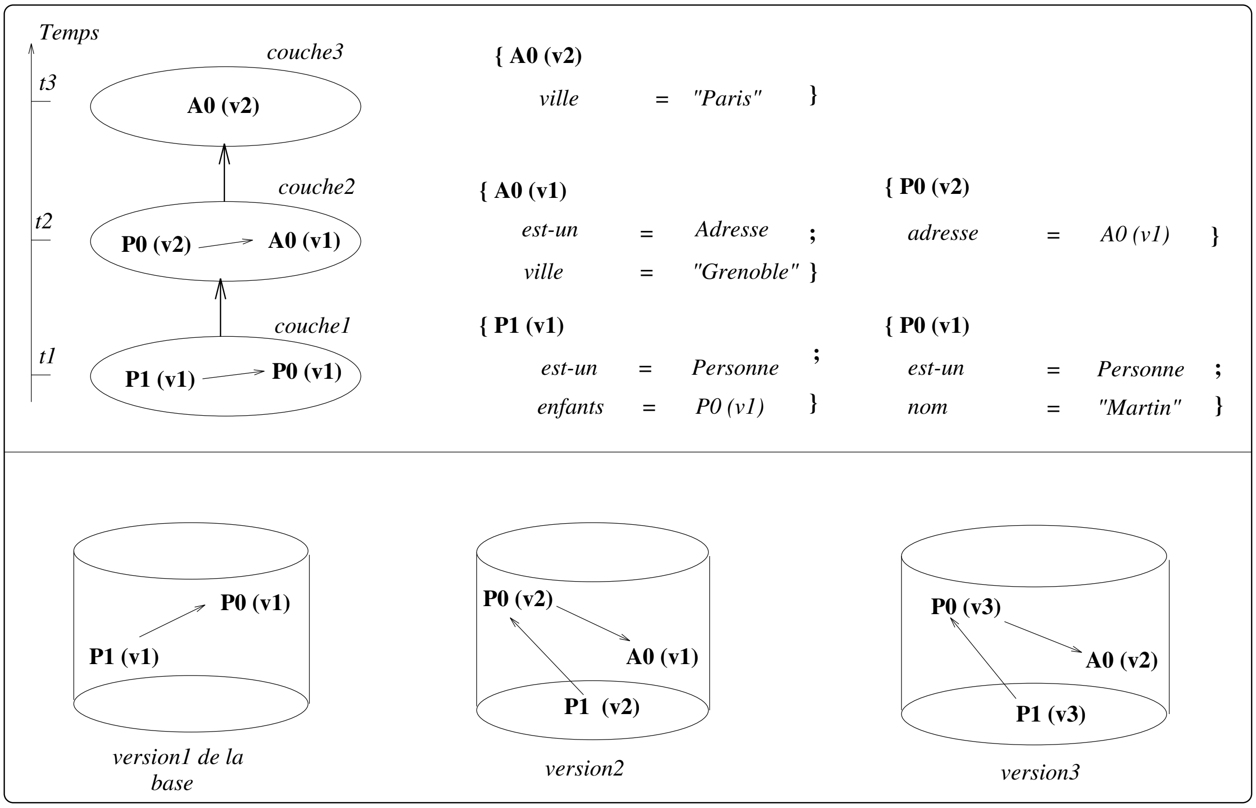


Figure 3.14 - : La partie supérieure de cette figure montre le contenu statique des couches lors de leur création. *Couche1* contient les deux instances $P0$ et $P1$. La deuxième pointe sur la première. L'instance $A0$ a été créée dans la *couche2*. Dans cette même couche $P0$ a donné la valeur $A0$ à son attribut *adresse*. $A0$ a ensuite été modifiée dans la troisième couche. La partie inférieure de la figure montre les trois versions de la base. Nous remarquons que $P0$ et $P1$ possèdent chacune une version logique dans chaque version de la base de sorte que la première version de $P1$ pointe bien sur la première version de $P0$ et la $2^{\text{ème}}$ sur la $2^{\text{ème}}$, etc. Ainsi, chaque version de la base est cohérente.

3.5.3 Opérations sur les classes

Nous nous sommes jusqu'à présent intéressés à l'évolution des instances. Dans ce paragraphe, nous allons nous intéresser à l'évolution possible des classes, notre système supportant également le versionnement des structures de données.

L'évolution de la description d'une classe est une opération délicate pouvant avoir des répercussions importantes sur le reste de la base. C'est un sujet de recherche en soi. Dans la littérature, la plupart des modèles de versions existants proposent seulement des versions d'instances. Les systèmes de bases de données (`Orion`, `Encore`, `Avance`), que nous avons exposés au cours du chapitre 2, gèrent les versions de classes et les versions d'instances. En génie logiciel par contre peu de travaux ont vu le jour : la thèse [Ahmed-Nacer, 1994] en est cependant un exemple.

En ce qui concerne les bases de connaissances, parmi les travaux qui ont étudié le problème de l'évolution de classes, nous pouvons cité ceux de Capponi [Capponi, 1992] et de Chevalier [Chevalier, 1994]. Ces deux études ont proposé des solutions déterminant le comportement de `Shirka` vis-à-vis de la modification d'une classe. Notre travail se place dans la continuation de ces travaux et consiste à introduire les versions pour contrôler l'évolution des classes.

La modification de la définition d'un schéma de classe déjà existant peut s'effectuer suivant plusieurs formes : ajout, suppression ou modification d'un attribut. La modification d'un attribut peut concerner son nom, sa facette, son type ou la méthode (si elle existe) calculant sa valeur[14]. Chacun de ces modifications implique la création d'une nouvelle version de la classe concernée.

Nous écartons de notre étude la faisabilité d'une telle opération de modification. Nous nous intéressons uniquement aux conséquences possibles sur la base qui sont de deux natures différentes : conséquences sur la hiérarchie de la classe modifiée et conséquences sur les instances de la classe modifiée. Pour chacune, nous allons détailler ce que doit contenir la couche courante $\boldsymbol{C}$.

La classe modifiée n'a pas à changer de place dans la hiérarchie

Pour ce cas, la modification de cette classe entraîne des modifications au niveau des listes d'attributs hérités dans son sous-graphe (par exemple le cas du changement du nom de l'attribut *no-carte* dans la définition de la classe **Etudiant** de la figure 3.5). Ceci dit, nous n'avons pas besoin de redéfinir ces sous-classes de nouveau dans la couche $\boldsymbol{C}$, car les changements effectués au niveau supérieur sont transmis au niveau inférieur par la voie de quasi-héritage. Seule la classe dont la structure a été changée doit être redéfinie dans $\boldsymbol{C}$.

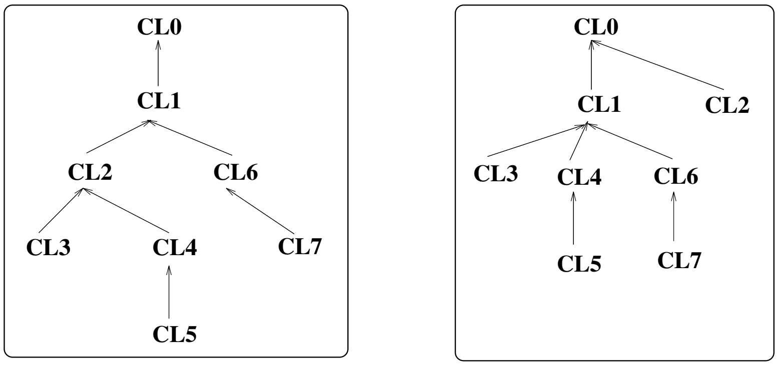
La classe modifiée doit être déplacée de façon à ce que sa hiérarchie initiale soit de nouveau correcte

Pour ce cas, les modifications apportées n'ont pas respecté la relation de spécialisation ou de généralisation (par exemple le cas de la suppression de l'attribut *âge* de la classe

[14] À part le fait de changer le corps de la méthode sans modifier son nom (qui est considéré comme un changement dans les valeurs des attributs d'une instance), toute autre sorte de modification (changement de nom d'une méthode existante, addition d'une nouvelle méthode) est considérée comme un changement dans la définition de la classe et implique la création d'une nouvelle version de celle-ci.

Etudiant de la figure 3.5). La classe modifiée peut être déplacée toute seule ou avec tout son sous-graphe.

- la classe modifiée est déplacée toute seule : la répercussion de cette opération a des conséquences sur la sur-classe et les sous-classes de la classe concernée. Illustrons cela par l'exemple de la figure 3.15. Il s'agit de la classe *CL2* qui a été modifiée et

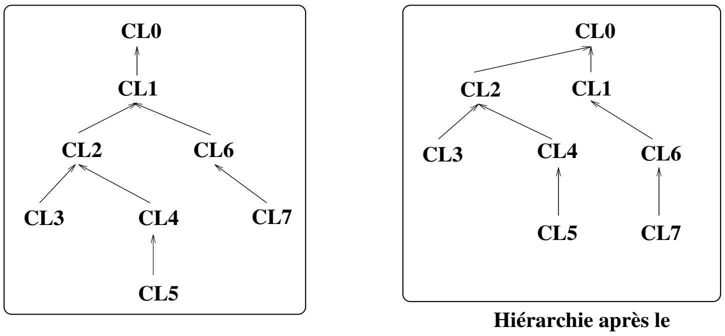


Hiérarchie avant la modification de CL2 **Hiérarchie après le déplacement de CL2**

Figure 3.15 - : La modification de la classe *CL2* a impliqué son déplacement. La classe *CL2* a été reinserée dans une nouvelle branche de sa hiérarchie.

doit changer de place. Les répercussions des modifications sur son graphe d'héritage initial concernent sa sur-classe *CL1* et ses sous-classes *CL3* et *CL4*. Les deux dernières vont être remontées d'un niveau pour avoir la première comme sur-classe. Ainsi, le système de gestion de versions va redéfinir la classe *CL1* (car sa liste de sous-classes a été changée) et les deux classes *CL3* et *CL4* (car leur sur-classe a été changée) dans la couche ***C***. Quant à la nouvelle branche (qui accueille *CL2*), le système insère la nouvelle sur-classe de *CL2* (ici *CL0*) dans la couche ***C***.

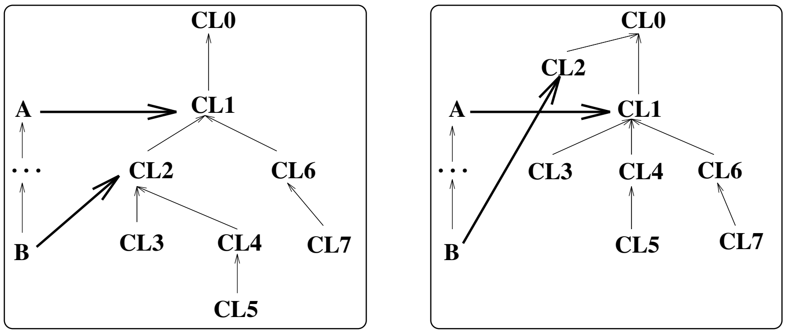
- la classe modifiée est déplacée avec tout son sous-graphe : la répercussion de cette opération a uniquement des conséquences sur la sur-classe de la classe concernée. Illustrons cela par l'exemple de la figure 3.16. Dans la couche ***C***, le système va



Hiérarchie avant la modification de CL2 **Hiérarchie après le déplacement du sous-graphe de CL2**

Figure 3.16 - : La modification de la classe *CL2* a impliqué le déplacement de tout son sous-graphe. La classe *CL2* ainsi que son sous-graphe ont été reinserés dans une nouvelle branche de la hiérarchie.

insérer la classe *CL1* et la classe *CL0* qui est la nouvelle sur-classe du sous-graphe concerné.



avant la modification de CL2 **après le déplacement de CL2**

Figure 3.17 - : Dans la hiérarchie initiale (avant la modification de *CL2*), la classe *A* référence *CL1* et sa sous-classe *B* référence *CL2* (qui est une sous-classe de *CL1*). Lors du déplacement de *CL2* dans une nouvelle branche (à cause de sa modification), une incohérence est signalée dans la hiérarchie des classes *A* ... *B* puisque la première pointe sur *CL1* qui n'est plus maintenant une sur-classe de *CL2* (référencée par *B*).

Le déplacement d'une classe (toute seule ou avec son sous-graphe) a également des conséquences sur les classes qui référencent les classes déplacées [Crampé, 1995]. La problématique est illustrée figure 3.17. La hiérarchie *A* ... *B* n'est plus cohérente après le déplacement de *CL2* puisque *A* pointe sur *CL1* et *B* (sous-classe de *A*) pointe sur *CL2* (qui n'est plus une sous-classe de *CL1*). Dans ce cas, deux solutions sont possibles dans `Shirka` : soit (1) changer les valeurs des références dans la hiérarchie qui référence la classe déplacée (par exemple dans figure 3.17 *A* va référencer la classe racine de la hiérarchie *CL0*. De cette façon, la hiérarchie *A* ... *B* respecte de nouveau les liens de spécialisation) soit (2) déplacer les classes dans la hiérarchie qui référence la classe déplacée (pour notre exemple, il s'agit de reclasser la classe *A* ou *B*). Le traitement de ces deux solutions nous ramène aux deux cas décrits ci-dessus (la classe modifiée n'a pas à changer de place, la classe modifiée doit être déplacée).

Les instances de la classe modifiée

Toute sorte de modification dans la définition d'une classe engendre des modifications sur les instances, puisqu'une instance ne peut exister hors de sa classe d'appartenance. Ses valeurs correspondent aux attributs dans la description de sa classe. `Shirka` impose la suppression de toutes les instances de chaque classe dont la structure (la liste de ses attributs) a été modifiée. La raison d'une telle décision, très stricte, est que `Shirka` n'a pas des mécanismes de vérification de type lui permettant de rattacher l'instance à sa nouvelle classe. Avec le support de versionnement, ceci a pour conséquence que la nouvelle version de la classe perd les instances de sa précédente version. Les anciennes instances ne seront donc ni converties, ni filtrées. Elles gardent néanmoins leur appartenance à l'ancienne version de la classe concernée. Les mesures à prendre au niveau de $\boldsymbol{C}$ sont donc celles présentées lors de la suppression des instances.

Cependant, conserver les instances des classes modifiées est une nécessité. Le sujet de thèse de Crampé [Crampé, 1995] concerne, entre autre, la mise à jour de ces instances conformément à la nouvelle définition de leurs classes modifiées. Nous avons déjà montré lors de l'étude de cette solution (Cf. §2.7.2) que la conversion ne permet pas de garder les anciennes instances (avant la conversion). Concernant notre problématique, il est indispensable que l'ensemble des instances d'une classe soit partagé par ses différentes versions. Pour ce faire, il faut utiliser des techniques comme le filtrage (Cf. §2.7.2). Cette proposition est une perspective nécessaire de cette partie de travail (concernant les versions des classes).

La plupart des autres systèmes refusent de perdre leurs instances (i.e. perdre des informations). Cependant, ils essayent de restreindre l'évolution structurelle. Par exemple, le système `ConceptBase` [Jarke et al., 1993] refuse toute modification dans la description d'une classe qui cause la violation des contraintes d'intégrité du modèle. Ainsi, toute modification nécessitant la migration des instances de la classe concernée est rejetée. Dans `Orion` [Kim et Chou, 1988], le domaine d'un attribut d'une classe ne peut que s'étendre. De cette façon, les instances de la classe n'ont pas besoin d'être reclassées et restent attachées à leur classe d'origine. De même, la suppression d'un type entraîne la suppression de toutes ses instances et leurs instances composantes si ce type est composite. Pour `Encore` [Skarra et Zdonik, 1987] et `Iris` [Beech et Mahbod, 1988], il est interdit de supprimer un type ayant des instances. Dans `Adèle3` [Ahmed-Nacer, 1994], les instances d'un type supprimé restent intactes, mais elles ne sont accessibles que pour la consultation.

3.6 Bilan et conclusion

La solution que nous avons proposée permet de résoudre élégamment les problèmes que nous avons mis en évidence au cours de la conclusion de l'état de l'art 2.8 :

1. Notre modèle gère les versions d'une base de connaissances. Toute entité définie dans la base (instance, classe, moyens d'inférence) profite automatiquement, et sans déclaration spéciale de la part de l'utilisateur, du support de ses versions. De plus, la gestion des versions d'une classe ou d'une instance est accomplie par le système de manière uniforme.

2. Le fait d'ajouter la fonctionnalité de versionnement aux connaissances (grâce à la gestion des versions des bases) n'a pas de conséquences sur le modèle de représentation de celles-ci. Ce modèle est resté intact.

3. Le mécanisme de quasi-héritage, défini afin d'élaborer les versions de la base, a résolu de manière naturelle les problèmes de référence et de configuration. C'est pourquoi, lorsqu'une entité (une instance ou une classe) est modifiée, sa nouvelle version est la seule à être insérée dans la couche courante. Toutes les autres entités qui la référencent (y compris les sous-classes d'une classe) n'ont pas à être recopiées dans la couche courante. C'est grâce au quasi-héritage que la cohérence des versions est assurée lors d'accès à celles-ci.

4. Le support de deux types d'évolution structurelle et individuelle par le modèle `Shirka` et la gestion du versionnement de classes et d'instances par notre système, incite à mettre en évidence quelques aspects essentiels. Chacun de ses aspects est précédé par un scénario permettant de l'illustrer :

   - Supposons qu'une instance $\boldsymbol{I}$ de la classe **Personne** (illustrée dans fig. 3.5) a été spécialisée dans la classe **Etudiant** dans la première version de la base. Puis à l'instant t (correspondant à une autre version postérieure de la base) $\boldsymbol{I}$ a eu son diplôme. En lançant le processus de classification, elle a trouvé une nouvelle classe d'attachement (la classe **Etudiant-diplômé**). Enfin, dans la version courante, $\boldsymbol{I}$ est devenue instance de la classe **Salarié** car elle a trouvé du travail. Nous arrivons alors à mémoriser la séquence d'événements produits pour une entité donnée tout au long de sa durée de vie (dans la base).

- Supposons un autre scénario où ***I*** ait deux classes possibles (pour la recevoir) à l'issue de sa classification. Ces deux classes sont stockées dans deux versions concurrentes. C'est une connaissance importante en soi car l'utilisateur peut demander de rattacher ***I*** aux deux classes et comparer ces deux hypothèses. De même, il peut, plus tard, confronter les séquences d'événements ou les historiques de l'évolution de ***I*** dans ces deux axes de travail.

3.7 Discussion

Après avoir présenté notre approche, nous allons au cours de cette section la comparer avec celle proposée par Cellary et Jomier [Cellary et Jomier, 1990]. Les deux approches gèrent des versions complètes d'une base. Cependant, il existe des différences nettes entre les deux solutions. Elles sont parfois techniques ou bien sont héritées d'une différence initiale des besoins et des objectifs fixés. Ainsi, nous commençons par citer les points communs. Nous détaillons ensuite les points de divergence.

3.7.1 Les points communs

L'approche des Versions de Bases de Données (VBD) semble similaire à la nôtre en ce qui concerne les points cités ci-dessous. Ceux-ci permettent, entre autres, de résumer les caractéristiques de notre modèle.

- La séparation entre le niveau physique et le niveau logique est maintenue dans les deux approches. Le mode de stockage ainsi que la représentation interne des versions sont ignorés de l'utilisateur; de la base il ne perçoit que l'ensemble de ses versions.

- Une version de la base est un ensemble de versions d'entités (une par entité) cohérentes entre elles. Cette cohérence est gérée automatiquement par le système. D'un autre côté, la création d'une version logique de la base entraîne la création logique de nouvelles versions de chacune de ses entités. Ainsi, au sein de la même version de la base, le problème de référence est résolu. C'est pourquoi nous parlons d'orthogonalité entre versions et liens de références (ou de composition).

- La relation de partage définie entre les versions permet à une version, dans les deux cas, de ne contenir que les informations qui ont été modifiées, ajoutées ou supprimées par rapport à sa version précédente. Par conséquent, plusieurs versions logiques de la base peuvent avoir des entités en commun et donc partager la même version physique.

Si l'intersection avec l'approche de VBD n'est pas vide, certains points de divergence ont été dégagés, montrant l'insuffisance de cette approche au regard de notre problématique.

3.7.2 Les points de divergence

La différence entre les objectifs fixés pour les deux approches a orienté les solutions proposées de manière à répondre à ces exigences. Les trois différences majeures sont les suivantes :

- Le cadre de travail n'est pas le même. L'approche VBD est dédiée aux bases de données, tandis que la nôtre vise les bases de connaissances. De nouvelles connaissances peuvent être déduites de celles déjà existantes dans la base. Cette dynamicité doit être prise en compte dans la mise en œuvre du système de versions. Ainsi, des mesures additionnelles (par rapport à l'approche de VBD) ont été nécessaires pour intégrer l'impact des opérations, comme la classification, sur la gestion de la cohérence de versions dans notre système.

- La sémantique est temporelle dans notre cas et elle est propre à l'utilisateur dans le cas des VBD. C'est pourquoi, une version est créée automatiquement par notre système chaque fois qu'une modification est apportée à sa base. L'ensemble des versions de la base représente l'historique de l'évolution de celle-ci. Dans l'autre approche, chaque version est générée explicitement par l'utilisateur. Contrairement (à nous) les anciennes versions ne sont pas figées; n'importe quelle version peut être modifiée. La suppression d'une entité est autorisée. Sa trace est par conséquent perdue puisque toutes ses versions sont également supprimées.

  Afin de profiter de cette sémantique temporelle, et pouvoir exploiter les connaissances historiques, le besoin d'un langage de requêtes adapté est crucial. Ainsi, nous avons été amenés à développer un tel langage (qui est décrit dans 5.3).

- Le but des deux études est différent et les techniques d'implantation sont par conséquent différentes. Pour [Cellary et Jomier, 1990], il s'agit de replacer une version d'entité dans son contexte. Les bases de données ont été choisies (comme moyens) pour servir de contexte. Quant à nous, notre objectif est de gérer des versions de la base de connaissances, ce qui nous a conduits à proposer, dès le début, une solution orientée-base et non pas une solution orientée-entité. Les couches, que nous avons définies, associent directement chaque version de la base à l'ensemble des versions d'entités qu'elle contient, ceci rend évident la reconstitution d'une version de la base. Or, pour l'approche de VBD, chaque version d'une entité est associée à l'identificateur de la VBD dans laquelle elle se trouve. Toutefois, il est possible d'accéder à une version complète de la base mais le processus est beaucoup plus compliqué. Il consiste, pour chaque objet, à parcourir toutes ses versions afin de déterminer celles qui possèdent l'identificateur de la version de la base recherchée [Gançarski, 1994].

  Plus généralement, il est beaucoup plus facile de concevoir des fonctionnalités de manipulation des versions de la base (comparaison, différence, etc.) avec notre solution qu'avec celle de VBD.

Pour conclure, le versionnement est géré pour des fins qui ne sont pas les mêmes pour l'approche de [Cellary et Jomier, 1990] et pour la nôtre. Les applications utilisant ces deux systèmes sont différentes.

Chapitre 4

Intégration du gestionnaire de versions dans l'environnement de construction

Nous allons décrire au cours de ce chapitre l'environnement développé pour l'élaboration d'une base consensuelle (BCons). Nous montrerons en particulier le rôle indispensable joué par le module de versions (exposé dans le chapitre précédent) au sein de cet environnement.

4.1 Introduction

L'objectif que nous nous sommes fixé dans le cadre de notre étude est de permettre à une équipe de personnes de réunir leurs connaissances au sein d'une base consensuelle. Pour nous, une connaissance (une description de classe ou une instance donnée) est considérée comme consensuelle si l'une des deux conditions suivantes est vérifiée :

- *Correcte*: la majorité (i.e. plus de la moitié) des personnes impliquées dans la construction l'a jugée "digne" d'être insérée au sein de la BCons. Par "digne" nous entendons que la connaissance en question ne met pas en cause le contenu des bases de connaissances de chacune des personnes participant à son examen.

- *Possible*: la majorité des personnes impliquées dans la construction l'a jugée comme "probable" (i.e. probablement valable) et elle peut alors être acceptée à titre d'hypothèse au sein de la BCons.

L'environnement informatique que nous voulons mettre en place doit contenir deux types d'agents :

- les constructeurs humains[1], qui ont pour rôle l'acquisition des connaissances et la conception de leur propre base contenant ces connaissances. Ils ont aussi la charge de bâtir la base commune et d'assurer que son contenu soit consensuel.

- le système informatique, qui doit :

    1. gérer les relations entre les différents utilisateurs afin de leur permettre de coopérer pour obtenir le consensus voulu. Pour ce faire, il est nécessaire de

[1] Différents termes seront utilisés par la suite pour désigner ces constructeurs : chercheurs, utilisateurs, clients de l'environnement.

mettre en place des interactions analogues à celles existant lors du travail en groupe. Ces interactions sont particulièrement étudiées dans le domaine des collecticiels. Ceux-ci proposent des outils informatiques pour améliorer la productivité des réunions de travail [Ellis et al., 1991] ou aider à la rédaction en commun d'un document [Streitz et al., 1992]. Il est clair que le système informatique doit favoriser le travail réparti et coopératif. Dans le cadre des systèmes multi-agents, de véritables efforts sont consacrés à l'organisation de ce type de travail [Boissier, 1990] [Werner et Demazeau, 1992]. Ce point est également une des occupations essentielle d'un atelier de génie logiciel [Munch, 1993]. Plusieurs solutions ont été proposées (voir par exemple [Melo, 1993] qui a étudié le problème de la programmation coopérative).

2. gérer la BCons ainsi que veiller à sa cohérence.
3. permettre aux multiples utilisateurs de la BCons de pouvoir examiner son évolution au cours du temps en mettant à leur disposition les diverses étapes de sa construction.

L'architecture proposée ici pour l'environnement de construction prend en compte tous les besoins mentionnés ci-dessus. Elle a été imaginée dans son principe par Jérôme Euzenat [Euzenat, 1995]. Il s'agit d'un réseau de cellules de coopération. Ces cellules permettent à un chercheur d'accéder au réseau, de discuter avec des autres constructeurs et de soumettre ses propres connaissances. Dans ce chapitre, nous allons montrer coment nous avons spécialisé et adapté le fonctionnement de ces cellules à notre problématique.

Avant de décrire les cellules de coopération, nous allons au cours de la section suivante distinguer les trois types de bases de connaissances gérées par l'environnement.

4.2 Les types des bases de connaissances manipulées par l'environnement

L'environnement informatique doit permettre l'élaboration d'une base contenant les connaissances consensuelles. Or, le fait que la BCons soit construite avec le concours d'un grand nombre de chercheurs rend délicate voire impossible l'obtention de ce consensus du fait de la présence de points de vue divergents. Il n'est en effet pas simple de faire collaborer efficacement un grand nombre de personnes. Ces difficultés se trouvent accrues lorsque le domaine que l'on cherche à modéliser dans la BCons est vaste car chaque chercheur ne peut avoir qu'une vue partielle des problèmes mis en jeu.

Nous croyons donc que pour atteindre un consensus il est généralement nécessaire de passer par plusieurs étapes intermédiaires au cours desquelles le domaine étudié est divisé en une série de sous-domaines plus restreints. Cette structuration offre un double avantage; d'une part elle permet de minimiser le nombre des chercheurs concernés ce qui facilite la communication et, d'autre part les domaines ainsi délimités deviennent plus facilement traitables. Une fois qu'un consensus est atteint dans tout ou partie de ces sous-domaines, il devient plus aisé d'effectuer une synthèse des consensus partiels.

Dans le cadre des projets de séquençage des génomes, des groupes de chercheurs se sont spécialisés dans des organismes donnés par exemple : *E.coli* ou *la levure*. Mais l'objectif est d'arriver à déterminer des fonctionnements plus généraux, concernant par exemple la régulation de l'expression des protéines, en comparant différents organismes.

De manière à refléter ce mode de travail, nous avons décidé de considérer trois types de bases de connaissances (voir fig. 4.1):

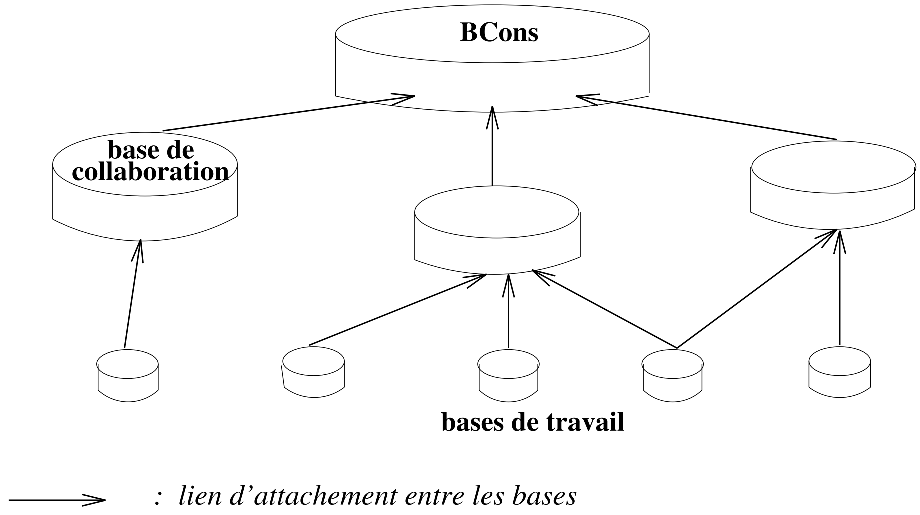


Figure 4.1 - : La hiérarchie des trois types de bases de connaissances manipulés par l'environnement de construction.

1. base de connaissances consensuelles;
2. base de collaboration;
3. base de travail.

Ces trois bases se différencient par leur contenu, par la façon de les construire, ainsi que par la manière dont leurs versions sont gérées.

Base consensuelle: elle représente la base construite par cet environnement. Elle contient les connaissances communes à une communauté de chercheurs sur un domaine particulier. La BCons est considérée comme une base publique dans le sens où son contenu est accessible en lecture à tous les utilisateurs autorisés par l'environnement. Son administrateur, qui est le responsable de sa gestion, est seul autorisé à la modifier.

Base de collaboration (BColl): la base de collaboration [2] correspond à la mémoire d'une équipe ou d'un projet scientifique travaillant sur un sous-domaine de la base consensuelle. Il existe donc autant de BColl que de sous-domaines traités. La BColl est construite de manière incrémentale par les membres de son projet et elle ne contient que les informations jugées correctes ou possibles par ceux-ci. L'accès à cette base est limité aux membres de son projet mais le responsable du projet est le seul habilité à la modifier. À chaque base de collaboration de l'environnement est associée au moins une base de travail qui participe à sa construction.

Base de travail (BT): elle correspond aux cahiers de laboratoire des chercheurs. Elle se trouve dans l'espace de travail de l'utilisateur. Elle peut être une copie de la base consensuelle ou de la base de collaboration dont elle fait partie. Elle peut également être bâtie petit à petit par l'utilisateur et contenir ainsi ses propres informations. Dans tous les cas, son propre utilisateur est le seul à y avoir accès.

[2]Cette base est aussi nommée base de groupe.

Cette organisation de bases de connaissances (BC) repose sur une notion parallèle à celle des espaces de travail proposés par [Katz, 1990] dans le sens où les droits d'accès sont distribués en fonction de chaque type de base.

La figure 4.1 montre les liens d'attachement entre les trois types de bases. Ces liens sont imposés par le protocole de construction adopté et qui est basé sur le processus de publication par soumission (voir 1.3). En pratique, afin de publier un article, son auteur le soumet à une revue ou une conférence en lien avec le sujet de recherche traité. L'article est ensuite jugé par des critiques qui peuvent refuser, accepter ou demander des modifications à l'auteur. De manière analogue, dans notre environnement les connaissances jouent le même rôle que l'article, l'auteur est le chercheur possédant ces connaissances, la conférence est la BC de groupe dont il participe à la construction et les juges sont les personnes de ce groupe (une étude détaillée du protocole de soumission est effectuée dans [Lemaire, 1993] et [Mahé, 1994]).

Par définition, une base A est attachée à une base B si, d'une part l'utilisateur de A participe à la construction de B par envoi (soumission) de nouvelles connaissances et d'autre part si, à chaque demande de modification de B, l'utilisateur de A participe avec les utilisateurs des autres bases attachées à B à l'examen de cette proposition et à la décision de mise à jour de la base. Les utilisateurs des BT participent à l'élaboration de la (ou des) BColl de leur équipe, tandis que les administrateurs des BColl soumettent les connaissances de leur base à la BCons correspondante.

Les bases de collaboration et de travail ont besoin d'être déclarées pour la première fois dans l'environnement par leur responsable de projet et leur utilisateur[3] respectifs. Cette déclaration est accomplie par une demande de la part du responsable.

Chaque base de l'environnement peut être vue comme une base consensuelle. La différence réside dans le nombre de personnes concernées par ce consensus. Les bases de collaboration représentent un consensus sur les connaissances traitées dans un projet de recherche. Elles contiennent donc les connaissances communes à toutes les bases de travail attachées. La base de travail est une base consensuelle *particulière* puisqu'elle contient toutes les connaissances avec lesquelles son utilisateur est en accord.

Remarque : la relation ensembliste entre l'ensemble des connaissances d'une BColl et celui de ses BT peut être exprimée par : $(\cap \text{ BT}) \subset \text{BColl}$. La raison pour laquelle cette relation est une inclusion et non pas une égalité est que la BColl peut accepter différentes hypothèses (axes de travail) qui peuvent n'appartenir qu'à un certain nombre des BT concernées.

Nous distinguons trois types de versions relatives à ces trois types de bases. Ils sont définis de la manière suivante :

- **Version publique :** elle est liée à la BCons. Les personnes abonnées sont les seules à avoir le droit d'y accéder pour la consulter. Elles sont averties à chaque fois qu'une nouvelle version publique est créée. De cette manière, elles peuvent mettre à jour leurs propres bases de connaissances (les BT).

- **Version de transition :** elle représente une version d'une BColl. Elle n'est consultée que par les utilisateurs des bases de travail qui sont attachées à cette base. Ces utilisateurs sont avertis à chaque fois qu'une nouvelle version de transition est créée.

[3] Un utilisateur d'une BT faisant partie de l'environnement est considéré abonné à celui-ci.

- **Version privée (ou de travail) :** elle n'est accessible pour consultation ou pour modification que par l'utilisateur qui l'a créée. Une version de transition (ou une version publique) devient une version de travail dès qu'elle est chargée dans l'espace de travail d'un utilisateur.

Le passage d'un type de version à un autre dépend des relations définies entre leur base respective (voir fig. 4.1). Une version de travail ne peut pas par exemple devenir directement une version publique, et être ainsi incorporée dans la BCons. Seules les versions de transition le peuvent sous la condition d'être examinées et approuvées par les chercheurs responsables de la BCons. Une version privée doit également être soumise[4] à une base de collaboration afin d'être validée par les membres du projet de celle-ci et devenir par conséquent une version de transition (appelée ainsi car elle représente une étape de transition entre la version privée et la version publique). Cette façon de procéder constitue nos hypothèses de travail et explique les interactions illustrées dans la figure 4.2. En résumé, une version publique est créée chaque fois qu'une version de transition est acceptée au sein de la BCons. Tandis qu'une version de transition dans une BColl est créée chaque fois que les responsables de cette dernière valident une version privée soumise par un utilisateur.

Avoir plusieurs étapes de transformation des versions n'est pas une idée nouvelle. Nous la rencontrons dans [Katz et Lehmann, 1984], [Chou et Kim, 1988] ou [Palisser, 1990]. Pour ces travaux, les versions passent d'un état (privé par exemple) à un autre état (public par exemple) en se basant sur des critères de stabilité. Une version est jugée stable lorsqu'elle n'a plus besoin d'être modifiée et qu'elle est figée. L'enchaînement entre l'état initial (non stable) d'une version et son état final (stable) constitue le cycle de vie de cette version. Dans notre cas, les versions, une fois jugées et intégrées dans leur base correspondante, sont considérées comme stables (i.e. figées), peu importe le type de leur base d'appartenance.

De manière à gérer les interactions entre les différents utilisateurs, l'accès aux différents types de bases de connaissances vus précédemment n'est pas réalisé directement mais à travers des cellules de coopération (voir fig. 4.3) mentionnées au cours de l'introduction de ce chapitre. Nous allons maintenant revenir plus en détail sur l'architecture de ces cellules et le fonctionnement que nous leur avons attribué dans le cadre de ce travail.

4.3 Description d'une cellule de coopération

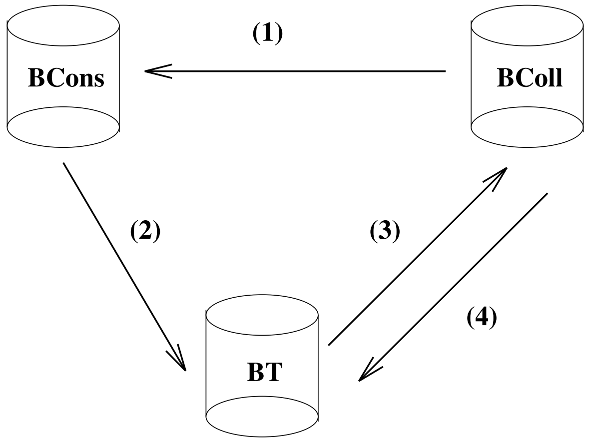
Une cellule de coopération est composée de six modules (voir fig. 4.4) :

- les modules chargés d'assurer le processus de construction de la BCons. Ce sont : le modules de gestion de la cohérence, le module de négociation et le module de gestion de versions.

- les modules chargés d'assurer le fonctionnement de la cellule et son intégration au sein de l'environnement. Ce sont : l'interface de gestion des interactions, le module de fonctionnement interne et le module de transport.

Une cellule contient également deux bases : une base de connaissances (d'un des trois types précédents) et une base de données, contenant toutes les informations nécessaires pour le bon fonctionnement du réseau. Elle est physiquement localisée au même endroit que la base de connaissances associée.

[4] Le protocole de soumission est donnée dans §4.4.3.



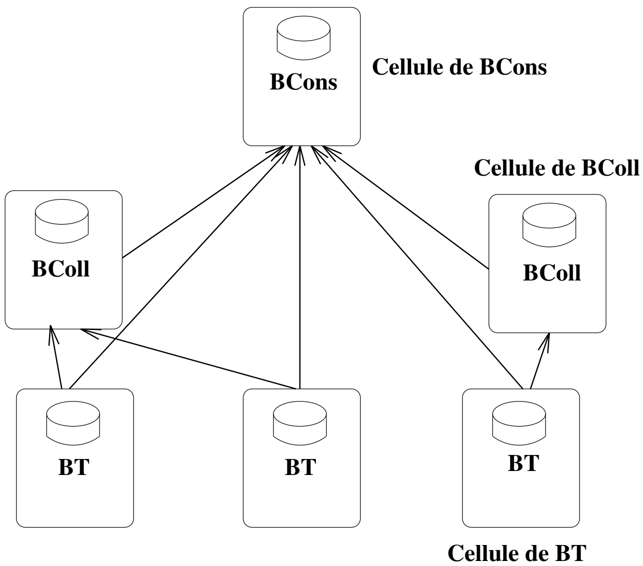
(1) : une version de transition est devenue une version publique après sa validation auprès des développeurs de la BCons. Or une version publique ne peut pas être chargée dans une BColl et ainsi devenir une version de transition. Une BColl est uniquement élaborée par les soumissions de ses chercheurs

(2) : une version publique est chargée dans la base de travail d'un utilisateur et est ainsi devenue une version privée

(3) : une version privée est devenue une version de transition car elle a été soumise à la BColl concernée et acceptée par les membres du projet de celle-ci

(4) : une version de transition est chargée dans l'espace de travail d'un utilisateur et est devenue sa version privée

Figure 4.2 - : Les relations d'échange entre les trois types de bases.



⟶ : *lien de connexion entre les différentes cellules*

Figure 4.3 - : L'accès aux différents types de bases de connaissances est réalisé à travers des cellules de coopération. Toutes les bases de l'environnement peuvent accéder à la BCons. Les BT accèdent de plus aux BColl auxquelles elles sont attachées.

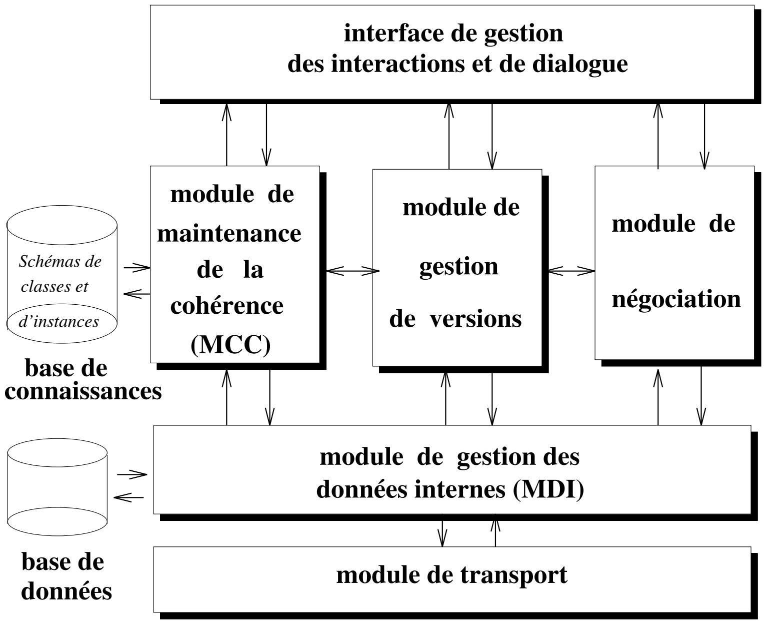


Figure 4.4 - : L'architecture d'une cellule de coopération.

Nous allons maintenant détailler les différents modules dans les sous-sections ci-dessus; le module de versions étant étudié en section 4.4.

4.3.1 Interface de gestion des interactions et de dialogue

Cette composante est le point d'entrée du réseau de coopération. Elle sert d'une part à accéder aux autres composants de sa cellule, et d'autre part à gérer les commandes de l'utilisateur. Celui-ci doit tout d'abord se connecter au réseau afin d'avoir accès à cette interface. Il a besoin pour cela d'entrer son nom et son mot de passe. Une fois la connexion établie, il a alors à sa disposition deux catégories de services : les services destinés à des bases extérieures et les services destinés aux bases locales :

- **les services extérieurs** sont :

  1. *Charger* une base de connaissances extérieure à la cellule de coopération considérée. Il faut préciser que lorsqu'un chercheur est connecté, ses bases locales (s'il possède plusieurs BC) ainsi que les BD correspondantes sont automatiquement chargées dans son espace de travail;
  2. *Interroger* à distance une base extérieure;
  3. *Soumettre* de nouvelles connaissances afin de contribuer à la construction des BC auxquelles sa propre base est attachée;
  4. *Établir* une discussion[5] avec des autres collaborateurs, qui peuvent être des membres du même projet ou bien d'autres clients du réseau, concernant des connaissances propres à l'utilisateur;
  5. *Configurer* une base privée; cette commande sera détaillée dans la section 4.4.2;

[5] Il existe différentes manières pour établir une discussion. Nous ne les évoquons pas dans cette thèse. Nous tenons simplement à préciser que la messagerie électronique en est une.

6. *Maintenir un lien* entre la base locale de la cellule et une base de groupe. C'est la commande qui permet à son demandeur de rattacher sa BT à une (ou plusieurs) BColl. En conséquence, sa base sera déclarée dans l'environnement de construction de la BCons.

- **les services locaux**, exécutés par le module de gestion de versions, sont :

  1. *Créer* la base de connaissances (Cf. §3.4.1). La base de données de la cellule associée est créée automatiquement;
  2. *Interroger* une des deux bases locales : la base de connaissances par le biais des mécanismes de sélection offerts par le gestionnaire de versions (Cf. §3.4.2) et la base de données par l'intermédiaire d'un langage de requêtes exposé au cours de la section 5.3.2;
  3. *Mettre à jour* la base de connaissances via une des opérations de modification portant sur les instances (Cf. §3.5.1) ou sur les classes (Cf. §3.5.3) de la base;
  4. *Sauvegarder* la base de connaissances. La base de données est alors également sauvegardée.

En plus de ces services, cette composante doit également contenir une zone permettant l'affichage des messages envoyés par l'environnement en réponse aux requêtes du client. Cela peut être, par exemple, un message d'erreur indiquant qu'une des adresses des collaborateurs n'est plus valide.

4.3.2 Module de maintien de la cohérence

Le module de maintien de la cohérence (MCC) (fig. 4.5) a pour charge de juger toute connaissance soumise à sa propre base de connaissances. Lorsque de nouvelles informations sont fournies (par le chercheur local de la cellule ou par un autre client du réseau), ce module utilise les connaissances déjà existantes dans sa base pour les critiquer. Cette étape d'examen peut être divisée en trois phases :

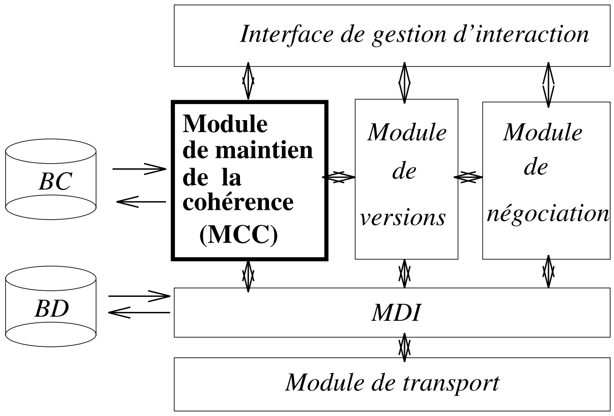


Figure 4.5 - : Le module de maintien de la cohérence au sein d'une cellule de coopération.

1. Identification : elle consiste à identifier les classes et les instances concernées par les nouvelles informations. On travaillera ainsi avec un sous-ensemble de la base au lieu de prendre en compte la totalité.

2. Vérification : il s'agit d'étudier les nouvelles informations dans le contexte résultant de la première phase et de faire des vérifications afin de décider de les accepter, les refuser ou les modifier. Ces vérifications s'effectuent sur les descriptions des entités de représentation (classes et instances), et concernent essentiellement la correction de la spécialisation et de l'instanciation. Pour ce faire, un système de types a été conçu [Capponi, 1995], qui calcule, sur la base des descriptions, un type (respectivement une valeur) pour chaque classe et attribut (resp. instance). De cette façon, l'instanciation correspond à l'appartenance d'une valeur à un type, et la spécialisation est vérifiée par une relation de sous-typage adaptée.

3. Adaptation : elle consiste à déterminer le comportement de la BC face aux nouvelles informations, selon le résultat de l'étape précédente. Trois cas possibles sont à distinguer :

   - Accepter les nouvelles informations dans la base. Cette décision est propagée au module de gestion de versions qui prend les mesures nécessaires. Supposons par exemple qu'une demande de modification de la définition d'une classe soit faite auprès d'une BC. La cellule de celle-ci va diriger cette requête au MCC. Si le résultat est positif, ce dernier précise alors s'il faut ou non changer la place de la classe concernée dans la hiérarchie. C'est alors au gestionnaire de versions d'intervenir. Selon la description donnée au cours de §3.5, une nouvelle couche sera créée afin d'incorporer les conséquences d'une telle modification.
   - Rejeter les nouvelles informations en expliquant la raison pour laquelle cette décision a été prise (ex: cause de la contradiction, etc.). L'explication est fournie par le système et renvoyée immédiatement au chercheur concerné à travers la composante graphique. Aucune action n'est effectuée au niveau des couches par le système de versions.
   - Accepter les nouvelles informations à titre d'hypothèse. Un nouvel axe est alors créé par le gestionnaire de versions (Cf. §3.4.1).

Remarque : ce module ne peut pas accomplir sa tâche de validation de nouvelles connaissances, sans supposer les deux contraintes suivantes :

1. l'unicité du formalisme de représentation de connaissances pour toutes les bases (nous travaillons ainsi avec des bases dites homogènes) aux niveaux syntaxique et sémantique. Le niveau syntaxique détermine le langage de description des différentes connaissances. Le niveau sémantique donne une signification dans le monde réel aux éléments de représentation [Mariño, 1993].

2. l'utilisation de la même terminologie par tous les chercheurs. Ainsi, il n'est pas possible d'avoir dans les bases privées des chercheurs deux entités différentes ayant la même syntaxe et la même sémantique (par exemple, l'entité *Personne* est décrite par son nom et son âge et l'entité *Individu* est définie de la même façon). Pareillement, il est interdit d'avoir une même entité décrite différemment par les chercheurs (par exemple, l'entité *Personne* qui est décrite par un chercheur par son nom et son âge et par un autre elle est décrite par son prénom et sa date de naissance).

4.3.3 Module de négociation

Dans un cadre de travail collectif, [Barbian et Schlageter, 1993] distinguent trois niveaux d'interactions entre les personnes impliquées :

- Communication : il s'agit de l'activité d'échange de messages entre les agents. Ceux-ci n'ont pas forcément un but commun ou ne partagent pas les mêmes données.
- Coordination : il s'agit d'harmoniser le travail entre les divers membres d'un groupe à la poursuite d'un but commun.
- Collaboration : il s'agit de synchroniser l'accès et le partage d'une même ressource pour l'obtention des données.

Le but principal du module de négociation (fig. 4.6) est de permettre aux différents chercheurs de communiquer, de se coordonner et de collaborer en favorisant toutes sortes d'interactions entre eux.

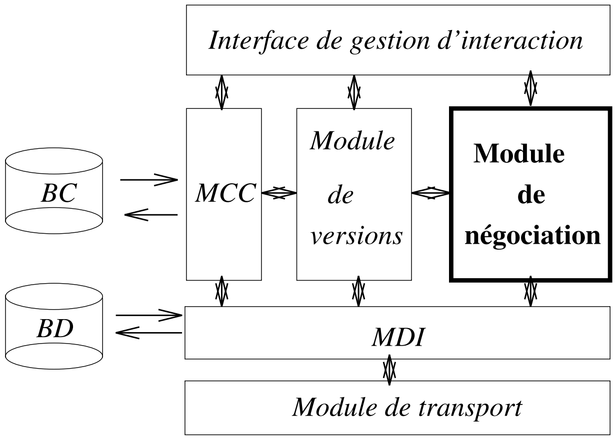


Figure 4.6 - : Le module de négociation au sein d'une cellule de coopération.

Les chercheurs ont besoin de communiquer afin de

- résoudre un problème ou un conflit. En effet, même si les chercheurs coopèrent volontairement, des conflits peuvent apparaître soulignant un désaccord ou des points de vue différents. Établir un dialogue peut résoudre ce type de conflits.
- mettre à jour les autres BC. Afin que les soumissions d'un chercheur puissent être acceptées, il est indiqué de demander conseil auprès des autres personnes concernées par ses bases.
- sous-traiter le travail. la situation donnant naissance à ce type d'interactions est l'incapacité d'un chercheur à pouvoir répondre localement au problème posé. Cette incapacité peut avoir des origines diverses (manque de compétence sur le domaine, manque de disponibilité, etc.). Un exemple typique est celui de la validation de nouvelles connaissances. Nous pouvons imaginer dans ce cas que le chercheur responsable de cette décision peut demander à ses collègues de l'aider à formuler sa décision. Il peut également choisir de déléguer totalement sa responsabilité à une autre chercheur dont il connaît les compétences.

Dans ce cas, le module de négociation va fournir au module de gestion des données internes (MDI) les informations à communiquer ainsi que les adresses des personnes à contacter. Réciproquement, les réponses des cellules interrogées parviendront au module via le MDI.

Les chercheurs ont besoin de se coordonner afin de

- juger des connaissances soumises à leur base commune.
- juger une demande d'attachement d'une base à leur base. À titre d'exemple, supposons qu'une personne demande à faire partie d'une équipe de recherche. À ce moment, tous les chercheurs concernés par la BColl de cette équipe doivent voter afin d'accepter ou non cette demande (i.e. d'établir ou non un lien entre la BT du demandeur et leur BColl).

Le rôle du module de négociation permettant de gérer ce type d'activité (le jugement) est détaillé dans la section 4.4.4.

Les chercheurs ont besoin de collaborer afin de résoudre les conflits d'accès concurrents à leur base commune. Il est à noter que le nombre d'accès dépend du nombre de connexions autorisées par la machine (où réside la base concernée). Plusieurs utilisateurs peuvent travailler simultanément sur les différentes versions de la même base sans se gêner. Le problème se pose dans le cas des accès concurrents sur la même version. Le module de négociation impose les contraintes de synchronisation suivantes :

- le mode d'écriture est réservé aux responsables de gestion de chaque base. Ils ne sont pas concurrents car il existe un responsable par base de connaissances. Cependant, aucun responsable ne peut écrire lorsqu'une demande de lecture (ou de soumission) est en train d'être exécutée.
- plusieurs lectures peuvent être réalisées en même temps. Par contre, aucune lecture n'est autorisée pendant une opération d'écriture.
- lors d'une demande de soumission, tout autre accès, hormis la lecture, à la version concernée est interdit tout au long du processus de soumission. Dans le cas où les connaissances à insérer ont été refusées, l'accès est de nouveau permis. Sinon, il faut attendre la fin de l'étape d'intégration.

Le module de négociation peut également définir un ordre de priorité entre les trois opérations (par exemple : priorité à l'écriture, puis à la soumission et enfin à la lecture). La façon de gérer le processus d'accès et les priorités ne rentre pas dans le cadre de cette thèse.

4.3.4 Module de gestion des données internes

Le rôle du module de gestion des données internes (MDI) (fig. 4.7) est de maintenir au sein de la base de données l'ensemble des informations permettant d'assurer le fonctionnement de sa cellule. Le contenu d'une base de données dépend du type de la base de connaissances associée.

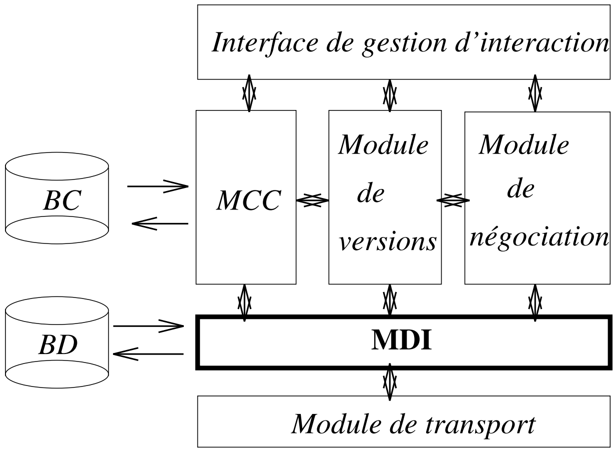


Figure 4.7 - : Le module de gestion des données internes au sein d'une cellule de coopération.

La BD dans le cas d'une BT contient les informations suivantes :

1. le nom de la BT fourni par son propriétaire (l'utilisateur);
2. le nom de son propriétaire ainsi que son mot de passe (sécurité d'accès pour la protection des données);
3. le numéro de la version courante (par défaut il s'agit de la version la plus récente) ainsi que sa date de création. Celle-ci indique la dernière opération de mise à jour effectuée puisque la création d'une version correspond à la mise à jour de sa BC;
4. le nom et l'adresse électronique du serveur de la cellule concernée.
5. les noms des BColl auxquelles la BT est attachée. Le système associe à chaque nom l'identité de sa cellule de coopération (il s'agit de l'adresse électronique de son serveur);
6. le nom et l'adresse électronique du serveur de la BCons de l'environnement;
7. les adresses électroniques de tous les abonnés aux mêmes BColl de la BT en question. Ces adresses sont collectées dans les bases de données de chacune de ces BColl. Leur utilité sera justifiée lors de l'étude des interactions entre les différents modules d'une cellule.

La BD dans le cas d'une BColl contient comprend les données suivantes :

1. le nom de sa BColl;
2. le nom et le mot de passe du responsable administratif gérant sa BColl. Il s'agit de la seule personne ayant le droit d'écrire et modifier celle-ci;
3. les adresses électroniques de tous les utilisateurs abonnés à la BColl (dont la BT est attachée à la BColl). Ces adresses servent, entre autres, à déclarer les droits d'accès à la base;
4. le nom du projet et le domaine de recherche de la BColl;

5. le numéro et la date de création de la version courante. Cette date correspond également à la date de la dernière opération de modification effectuée;
6. le nom et l'adresse électronique du serveur de la cellule concernée;
7. le nom et l'adresse électronique du serveur de la BCons à laquelle la BColl en question est rattachée.

Lors de la création d'une BColl par son administrateur, ce dernier initialise la BD correspondante. En cours d'utilisation, lorsqu'une BT est attachée à une BColl, le module de gestion de cette dernière ajoute à la BD l'adresse électronique de l'utilisateur de la base de travail.

La BD dans le cas de la BCons contient les informations suivantes :

1. le nom de sa BCons;
2. le nom et le mot de passe de l'administrateur de sa BCons;
3. les adresses électroniques de toutes les personnes autorisées à accéder à cet environnement;
4. le nom du domaine de recherche couvert par la BCons;
5. le numéro et la date de création de la dernière version publique (qui est la date de la dernière opération de mise à jour de la BCons);
6. le nom et l'adresse électronique du serveur de la cellule concernée;

Cette BD est également gérée par le responsable de sa BCons. Son contenu est modifié lorsqu'une nouvelle BColl est rattachée ou détachée de sa BCons, lorsque cette dernière est modifiée ou enfin lorsqu'une nouvelle BT est ajoutée ou retirée de l'environnement.

4.3.5 Module de transport

Ce module est la composante permettant la connexion entre les multiples cellules du réseau. Une cellule est connectée à une autre si leurs deux machines respectives sont connectées via le réseau *Internet*. La tâche principale de ce module est de transmettre les messages (les requêtes ou les réponses) de la cellule source à la (ou les) cellule(s) destinataire(s). Les points d'entrée et de sortie de ce module sont d'une part le MDI et d'autre part les modules de transport des autres cellules. La figure 4.8 est un exemple de fonctionnement de ce module.

4.4 Tâches du module de gestion de versions dans l'environnement

Nous allons uniquement nous intéresser au cours de cette section aux tâches relatives à l'intégration du module de versions au sein des cellules. Les opérations classiques de gestion de versions ont déjà été décrites au cours du chapitre 3.

Il s'agit de deux catégories de tâches. D'une part les tâches associées à la mise en place des connaissances initiales (chargement et configuration). D'autre part, les tâches associées au partage de connaissances (soumission, négociation et intégration).

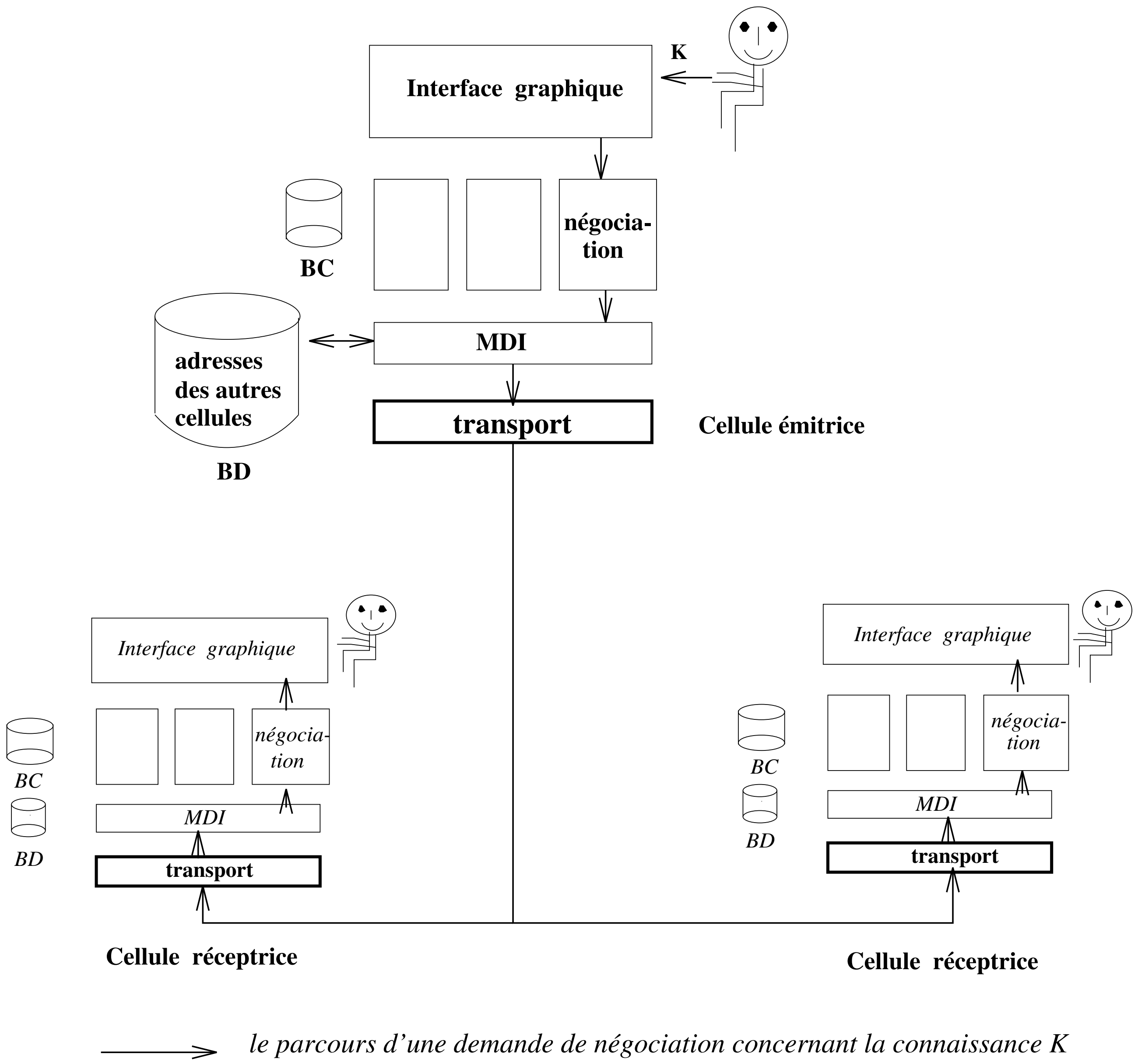


Figure 4.8 - : Un exemple de fonctionnement du MDI et du module de transport. Le chercheur possédant une connaissance K, veut l'envoyer à ses collègues. Le MDI va consulter la BD de sa cellule afin d'extraire les adresses des cellules concernées par la demande du chercheur. Le module de transport va ensuite utiliser ces adresses afin de transmettre la connaissance K aux chercheurs des cellules concernées.

4.4.1 Chargement

L'utilisateur peut demander la construction de sa propre BT en réutilisant les connaissances présentent dans la BCons ou les BColl auxquelles il est attaché. La tâche de chargement permet dans ce but de récupérer une (ou plusieurs) version(s) déjà validée(s) d'une base existante.

Mécanisme de communication et de mise en œuvre

Ce mécanisme est illustré figure 4.9. L'utilisateur demande une requête de chargement (étape no.1 dans figure 4.9). Cette requête est transmise (2) au gestionnaire local de versions. Ce dernier va demander (3) à la BD associée, via le MDI, si le client en question a le droit d'y accéder. Si la réponse renvoyée est négative, le refus est transmis (4) à la composante graphique afin d'informer l'utilisateur. Dans le cas contraire, le MDI extrait (5)l'adresse de la cellule destinaire et lui transmet la commande de chargement par l'intermédiaire du module de transport. À la réception, le contrôle de versions extrait (6) les données requises de sa BC et les renvoie (7) à l'utilisateur concerné. Ces données vont former la première couche de la BT de cet utilisateur.

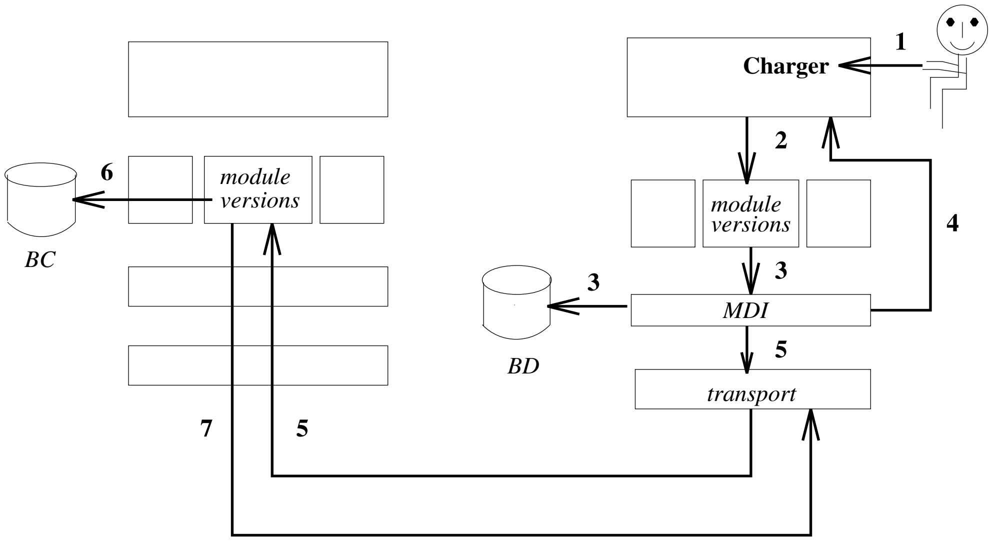


Figure 4.9 - : Le mécanisme de communication et de mise en œuvre de la tâche de chargement. Les chiffres montrent les différentes étapes d'exécution de cette tâche.

4.4.2 Configuration

Contrairement à la tâche de chargement, les connaissances récupérées sont sélectionnées dans les différentes versions d'une base existante. Un utilisateur configure[6] une base privée en choisissant les connaissances individuelles et structurelles qui répondent à ses critères. Il sélectionne des versions de classes et d'instances qui existent dans des bases publiques. Pour ce faire, il utilise les moyens de sélection exposés en § 3.4.2. Le mécanisme de communication est similaire à celui décrit lors de la tâche de chargement (Cf. §4.4.1).

Nous allons maintenant revenir sur le principe de configuration au travers d'un exemple.

[6] Cette opération est mise à la disposition des clients du réseau par le biais de la commande **Configurer** de l'interface graphique.

Exemple de configuration

Soit ***B1*** une BColl possédant dans son arbre de versions l'axe de travail décrit figure 4.10. Elle contient les connaissances sur la régulation de l'expression des protéines. La classe **opéron** décrit la structure générale d'un opéron (son mode de régulation et les différentes gènes qui le composent). Un membre du projet de cette BColl a besoin de développer une BC où figurent les opérons repressibles (i.e. appartenant à la classe **repression**) actuels et anciens (critère de la configuration). La version de transition la plus récente de ***B1*** lui donne les statuts courants des objets biologiques analysés dans son équipe. Si un objet biologique a été jugé repressible et puis a été analysé comme inductible (cas de *tryptophane*), il sera présent dans la version courante selon sa nouvelle présentation. C'est pourquoi, le chercheur en question a besoin de consulter les versions antérieures (des plus récentes aux plus anciennes) afin de trouver les informations recherchées. Cette recherche peut être manuelle mais peut également être automatisée grâce au gestionnaire de versions. Dans le cas de ***B1***, la réponse comprend les connaissances suivantes : les versions courantes de la classe **repression** et de l'instance *arginine* et la $2^{\text{ème}}$ version de l'instance *tryptophane*.

Après cette phase de sélection, le gestionnaire de versions de ***B1*** envoie les informations à son analogue de la cellule de l'utilisateur via le réseau. Le système procédera ensuite comme indiqué ci-après :

- création dans l'espace de travail de l'utilisateur d'une BC ainsi que de son arborescence contenant une seule branche avec un seul nœud représentant la première version privée;

- insertion dans cette couche les versions sélectionnées des connaissances désirées. De plus, les trois mesures suivantes sont prises par le système pour assurer la cohérence de cette couche :

  1. pour toute instance sélectionnée, le système insère sa classe d'appartenance dans la couche créée.
  2. toute version d'un schéma (classe ou instance) référencée par une classe ou une instance insérée y sera également ajoutée. Pour l'exemple précédent, il faut donc incorporer la version 3 de l'instance *r-arg* (référencée par la $3^{\text{ème}}$ version de l'instance *arginine*) et sa classe d'appartenance **gène-protéique**.
  3. pour toute classe insérée et qui ne représente pas une racine d'une hiérarchie, le gestionnaire doit également la compléter en lui ajoutant les attributs hérités de ses ascendants (qui ne sont pas été intégrés). Étant donné que c'est le cas des deux classes **repression** et **gène-protéique**, ces dernières vont avoir respectivement comme nouveaux attributs ceux de leurs sur-classes.

La première version de la base de l'utilisateur aura le contenu de la figure 4.11.

Remarque : en ce qui concerne la configuration, il est important de signaler les deux points suivants :

1. la version (d'instance ou de classe) récupérée pour configurer une base ne représente pas dans tous les cas des connaissances à jour. C'est le cas de l'instance *tryptophane* qui est insérée dans la base du chercheur comme étant opéron repressible, alors

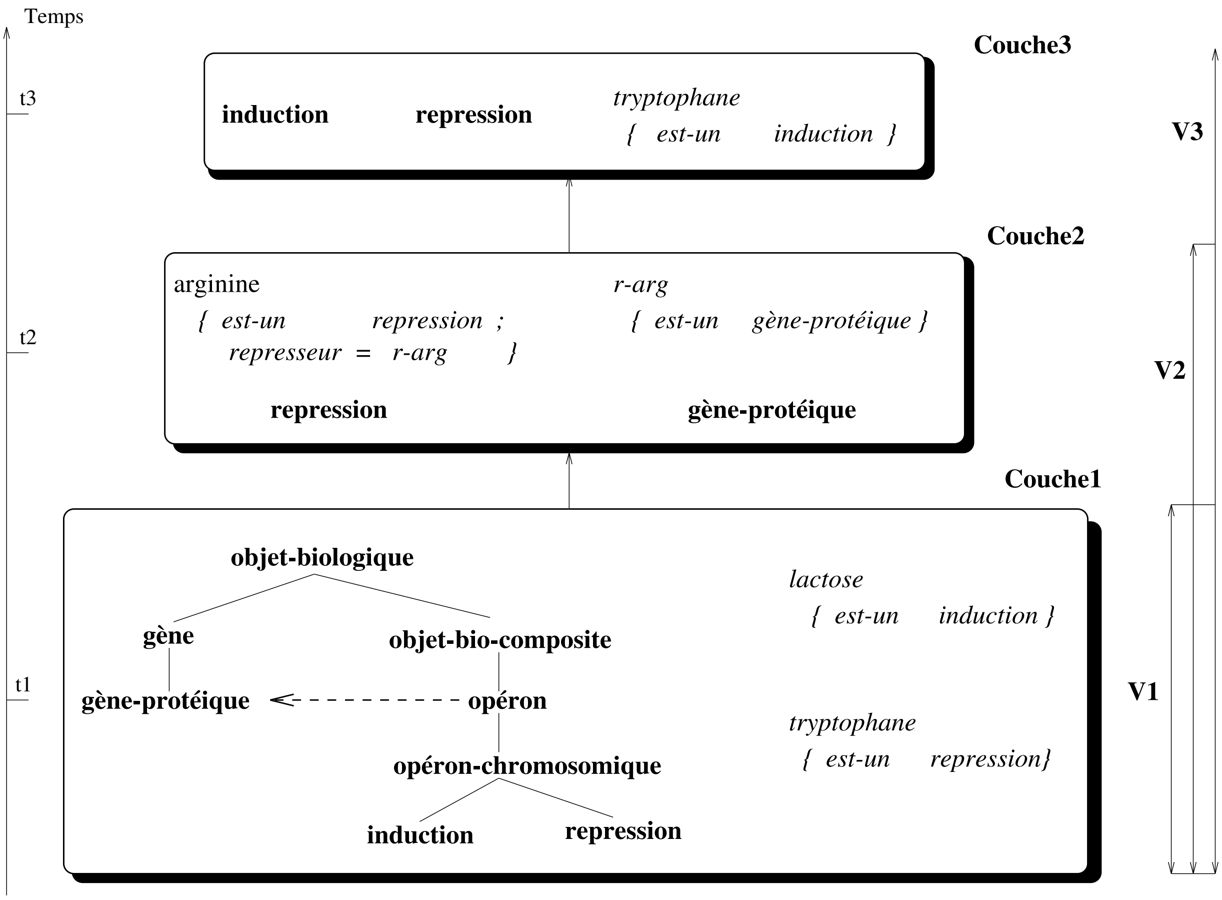


Figure 4.10 - : Exemple d'une base contenant des informations sur les objets biologiques. La classe *opéron* réfère la classe *gène-protéique* par l'intermédiaire de son attribut *represseur*. La première couche contient la hiérarchie initiale des classes de la base, ainsi que les deux instances : *lactose* de la classe **induction** et *tryptophane* de la classe **repression**. Les deux instances *arginine* et *r-arg* ainsi que leur classes respectives ont été ajoutées dans la couche2. Pour la couche3, l'instance *tryptophane* a changé de classe pour devenir instance de la classe **induction**.

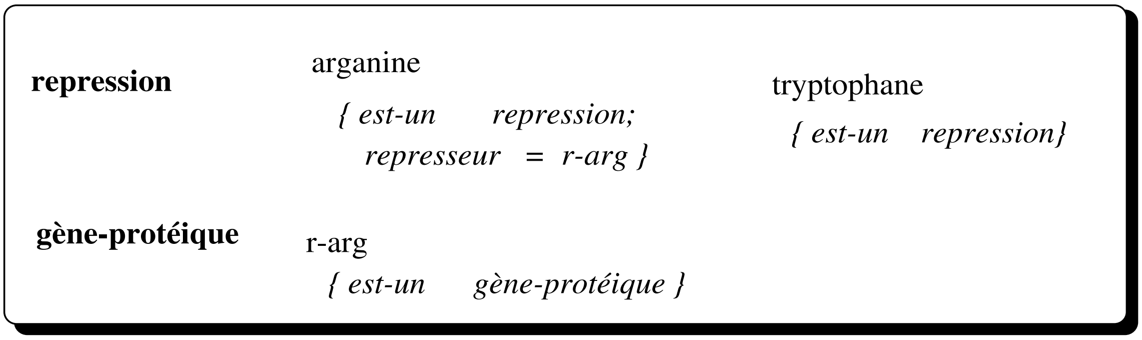


Figure 4.11 - : Une configuration possible de la base de connaissances de la figure précédente. Elle contient tous les opérons repressibles actuels et anciens.

qu'elle est en réalité (au moment présent) un opéron inductible (comme l'indique la version courante de la base d'où cette instance a été importée). L'utilisateur concerné doit ainsi être conscient de manipuler ces “anciennes” connaissances. Il peut toujours aller consulter leur base d'origine afin de connaître la date où ces connaissances ont été vraies (par exemple, *tryptophane* a été *repression* au moment *t2*). Il peut également demander cette information au système en lui indiquant le nom de la base d'origine.

2. l'opération de configuration n'est pas toujours possible à réaliser. Comme l'utilisateur peut demander, sans aucune restriction, de collecter des versions diverses de connaissances différentes, l'ensemble peut ne pas représenter un tout cohérent pour un noyau d'une BC. Il peut donc avoir une réponse négative (accompagnée d'une explication sur la cause de ce refus) à sa demande de configuration de la part du module de versionnement de sa cellule. Parmi les problèmes d'intégration des connaissances venant de bases multiples, tels qu'ils ont été exposés dans [Gambetta, 1993], nous ne faisons face qu'au problème de la cohérence de l'ensemble de ces connaissances. Les problèmes de syntaxe ou de terminologie ne peuvent pas apparaître dans notre cas où tous les abonnés à l'environnement informatique utilisent le même formalisme pour coder leurs connaissances et la même terminologie pour les décrire. Nous avons déjà mentionné cette hypothèse de travail.

Une fois le chargement ou la configuration terminé, le chercheur peut modifier sa propre copie de la base dans son espace de travail. Il est donc en mesure de faire les changements décrits en §3.5. Lorsque ceux-ci sont acceptés par le MCC local, ils sont incorporés dans une nouvelle couche créée par le gestionnaire de versions.

Parallèlement à ce travail réalisé localement pour construire sa propre base, l'utilisateur peut se servir du réseau afin de partager ses propres connaissances avec les autres clients du réseau. Pour ce faire, il doit franchir trois étapes : *soumission*, *négociation* et *intégration*. Dans les sections suivantes, nous décrivons les protocoles de celles-ci ainsi que le rôle joué par le gestionnaire de versions pour chaque protocole.

4.4.3 Protocole de soumission

Comme nous l'avons souligné auparavant, les connaissances sont soumises à une base en vue de leur intégration. Il faut noter que celles-ci ne peuvent pas être intégrées dans n'importe quelle base. La contrainte à respecter est celle de la hiérarchie existant entre les bases (voir fig. 4.1). Ainsi, lorsqu'un chercheur souhaite diffuser ses connaissances, il doit les envoyer à la BColl à laquelle sa BT est attachée tout en identifiant la version de transition qu'il veut mettre à jour. D'autre part, chaque fois qu'une nouvelle version de transition est créée, elle sera automatiquement soumise à la BCons selon un protocole identique.

Nous détaillons le cas de la soumission des connaissances entre une BT et une BColl. Soit ***B1*** la BColl concernée, et *V[i]* la version que le chercheur souhaite modifier (*i* étant son numéro). Cette version appartient à l'axe ***A1***. L'opération de soumission se déroule de la manière suivante :

1. le MDI de la cellule émettrice interroge sa base de données afin de trouver l'adresse de la base de connaissances réceptrice ***B1***.

2. le module de transport envoie l'ensemble des schémas à soumettre dans la couche choisie à l'adresse indiquée par le MDI.

3. le protocole de jugement est lancé par le module de négociation de la cellule destinataire afin de valider les connaissances dans la couche reçue. À partir de ce moment, aucun accès sauf consultation n'est permis à *V[i]*.

4.4.4 Protocole de négociation et de jugement

Les étapes de ce protocole sont :

- La première étape consiste à mettre à jour temporairement la version[*i*] de la base ***B1***. Le gestionnaire de versions va ainsi créer temporairement une nouvelle couche afin de recueillir les connaissances soumises. Cette couche sera positionnée au-dessus de la couche correspondante à la version[*i*] (i.e. couche[*i*]). Dans le cas où cette dernière n'est pas une couche feuille de ***A1***, le gestionnaire crée un nouvel axe de travail temporaire (voir fig. 4.12, cas2).

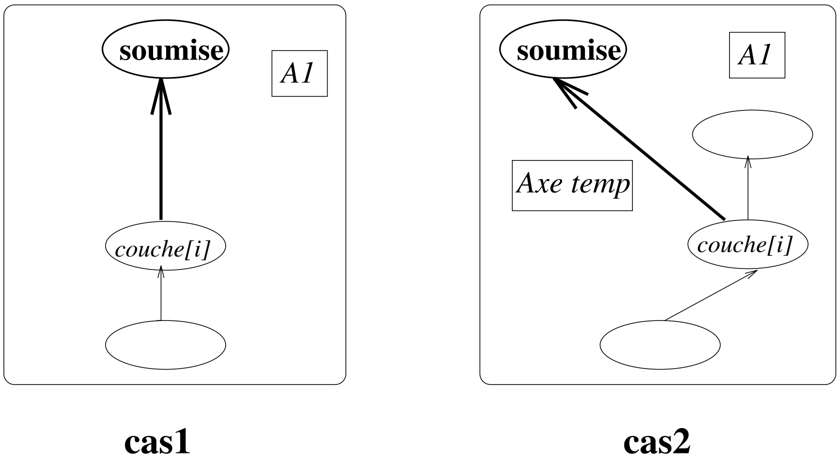


Figure 4.12 - : Ajout temporaire de la couche *soumise* afin d'être examinée au sein de la base ***B1***. Cette couche doit remplacer couche[*i*] si elle est acceptée. Elle est déclarée temporairement comme couche supérieure de couche[*i*]. Dans cas2 où cette dernière n'est pas une couche feuille, un nouvel axe, *axe temp*, est créé pour recevoir la couche *soumise*.

- Le MCC de ***B1*** va ensuite intervenir afin de déterminer si la couche ajoutée peut remplacer la précédente et devenir la couche courante. Autrement dit, il détermine si les schémas contenus dans la couche temporaire peuvent être incorporés dans la base sans pour autant remettre en cause la consistance de cette dernière. Cette phase de validation suit le schéma donné dans 4.3.2.

- Le module de négociation est ensuite activé. Il va jouer le rôle d'un médiateur dans le sens où il va désigner un certain nombre de critiques "humains" aptes à juger de nouveau les connaissances soumises. Son choix est basé sur l'axe de travail ***A1***. Pour cela, il va extraire de ***B1*** les adresses électroniques des clients supportant l'hypothèse en question[7]. ***B1*** étant une BColl du réseau alors les juges sont les utilisateurs des BT qui lui sont liées.

[7]chaque base de connaissances référence la liste des ses axes de travail. Chacun de ces axes possède la liste des adresses des chercheurs soutenant son hypothèse (voir 5.1.2).

- Les différents juges vont informer le module de négociation de ***B1*** de leurs commentaires et leurs notes concernant les connaissances à examiner ([Lemaire et Tayar, 1994] donnent plus de précisions sur la façon dont ces commentaires sont exprimés). Ce module va ensuite effectuer la synthèse des votes en considérant que la valeur finale est celle de la majorité des votes exprimés.
- La combinaison des résultats des juges humains et du système informatique produit le tableau 4.13 que nous allons commenter.

la synthèse des votes des experts humains	la réponse du module de révision de la base	la décision finale
accepter	accepter	intégrer les connaissances dans la base
rejeter	rejeter	refuser les connaissances
accepter	rejeter	réviser la base de façon qu'elle accepte la connaissance soumise
rejeter	accepter	refuser d'insérer les connaissances dans la base
accepter comme étant une hypothèse	accepter ou rejeter	créer une nouvelle branche dans la base pour incorporer les nouvelles connaissances

Figure 4.13 - : Le tableau de synthèse de votes. À chaque décision, est associée une action.

Dans les deux premiers cas, les deux camps prennent la même décision (accepter ou refuser), l'action sera identique (accepter ou refuser). Quant aux deux cas suivants, les deux avis ne sont pas les mêmes :

1. les critiques humains acceptent et le MCC de la base refuse. Cela peut se produire lorsque la base n'est pas complète[8] ou n'est pas à jour pour pouvoir accepter les connaissances suggérées. Le résultat final est l'incorporation de celles-ci dans la base sous réserve de modifier cette dernière[9].
2. les humains rejettent les changements proposés, par contre le système informatique les valide. Cela peut par exemple se produire lorsque les changements ne mettent pas en cause la consistance de la base mais qu'ils représentent en revanche une observation qui ne peut être produite selon les connaissances des juges humains. La décision finale est le rejet.

[8]Dans le cas d'absence d'information, c'est la théorie du *monde clos* qui sera appliquée. Cette théorie énonce que les connaissances qui ne sont pas déclarées comme vraies sont supposées fausses [Haton et al., 1991]

[9]Une solution possible est de supprimer dans la nouvelle couche créée les connaissances de ***B1*** qui ne sont pas cohérentes avec celles suggérées.

Lorsque la réponse du module responsable de la cohérence de la base est différente de celle de ses constructeurs, deux questions doivent être posées :

1. est ce que la base ***B1*** est toujours à jour? Les experts humains n'ont-ils pas des connaissances que la machine n'a pas? Ne faut-il pas réviser son contenu afin d'avoir le même comportement que celui des experts vis-à-vis de nouvelles soumissions?

2. le problème se trouve-t-il au niveau des experts humains eux-mêmes? Est-ce que leur refus n'est pas dû à des "a priori" de leur part (dans le cas où le rejet a été décidé, car les humains n'ont jamais rencontré une telle situation)?

La réponse à ces interrogations nécessite une discussion et un dialogue entre les chercheurs concernés. De toute façon, la décision finale prise par le module de négociation est celle des juges humains. L'avis primordial en ce qui concerne le contenu de la base est celui des spécialistes, notamment lorsqu'ils souhaitent le consensus de cette dernière. L'avis du module de révision sert en réalité à savoir si la base a besoin d'être mise à jour ou non.

4.4.5 Protocole d'intégration

L'intégration est la dernière étape. Elle est réalisée sous la responsabilité du module de versions. Elle est décidée dans trois situations distinctes (voir fig. 4.14) :

1. tous les avis sont en faveur de l'intégration. Le lien provisoire entre la couche[*i*] et la couche soumise est gardé comme lien permanent. La couche soumise devient alors la nouvelle couche courante (*cas1* et *cas2* de la figure 4.14). Pour *cas2*, il est demandé à l'utilisateur de fournir un nom pour le nouvel axe qui a été créé.

2. les experts humains sont d'accord et le système informatique est contre. Comme nous l'avons vu précédemment, les connaissances sont alors incorporées dans la base dès que celle-ci est apte à les recevoir (entre temps, ces connaissances sont gardées dans la couche temporaire). C'est pourquoi, les révisions à faire concernant le contenu de ***B1*** sont de nouveau décidées et votées par l'ensemble de ses chercheurs. Elles sont insérées dans une nouvelle couche créée par son gestionnaire de versions. La couche soumise est déclarée couche supérieure de celle-ci (*cas3* et *cas4* dans la fig. 4.14).

3. les chercheurs ont décidé d'accepter les connaissances soumises à titre d'hypothèses. La version associée à la couche soumise va être concurrente avec une autre version de l'axe ***A1***. Dans ce cas, les chercheurs doivent discuter pour pouvoir indiquer l'endroit exacte de l'insertion. Il s'agit de trouver une couche dans ***A1*** (autre que la couche[*i*]) qui soit en mesure d'accepter la couche soumise comme sa couche supérieure de façon à ce que la version résultante soit cohérente (*cas5* de la fig. 4.14). Si cela est impossible, la base se retrouve avec deux axes disjoints, l'ancien axe ***A1*** et le nouvel axe qui contient la couche soumise (*cas6* de la fig. 4.14).

Dans tous les cas, le gestionnaire de versions doit avertir les clients du réseau concernés par ***B1*** (dont les adresses se trouvent dans la BD de celle-ci) de l'existence d'une nouvelle version ou d'un nouvel axe de travail. Il doit aussi mettre à jour la base ***B1*** afin de changer le nombre total de ses versions, la date de création de sa version courante et d'ajouter, si besoin est, le nom du nouvel axe, l'hypothèse qui est à l'origine de son existence et le nom du client responsable de sa création.

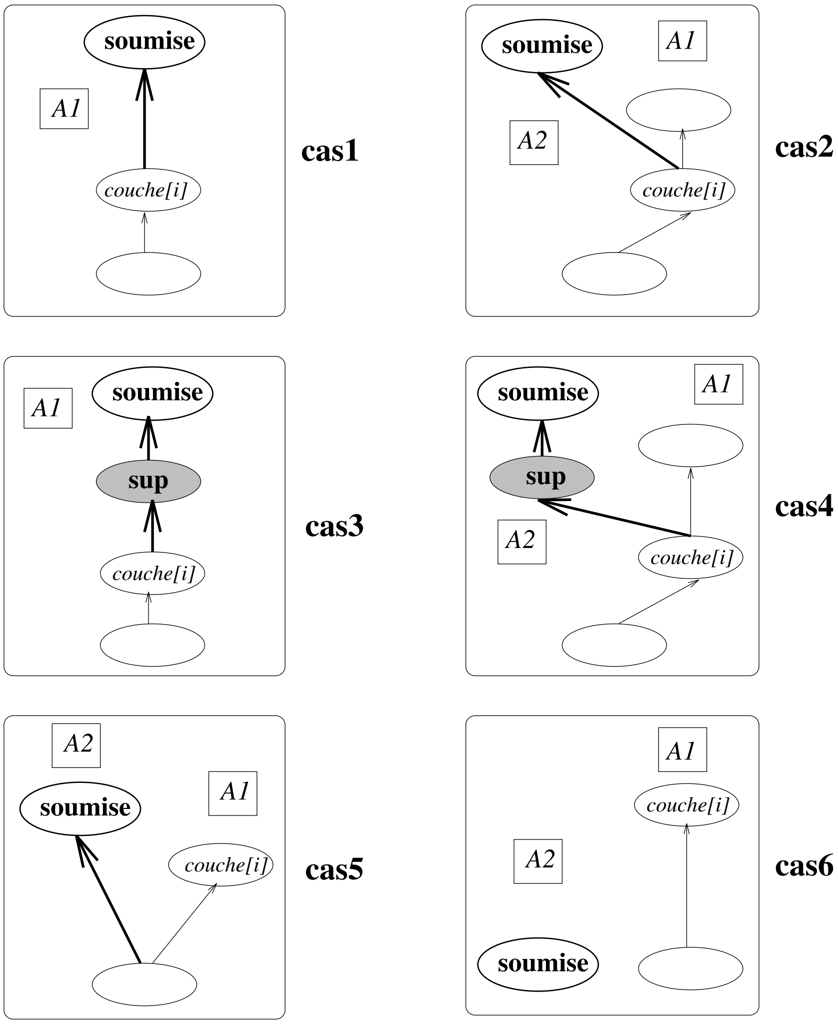


Figure 4.14 - : Les six cas rencontrés lors de l'intégration de la couche *soumise*. Dans cas3 et cas4, le système s'est trouvé obligé d'ajouter des couches supplémentaires *Sup* afin d'accepter *soumise*. Pour cas2, cas4, cas5 et cas6 un nouvel axe A2 a été ajouté.

Remarque : dans le cas où les connaissances soumises ont été refusées, le gestionnaire de versions va supprimer tout ce qui a été ajouté temporairement et la base concernée retrouvera son état précédent.

4.5 Conclusion

Au cours de ce chapitre nous avons tout d'abord présenté des trois types[10] de bases de connaissances ainsi que leurs types de versions correspondants. La distinction entre la base privée du chercheur et les autres bases de collaboration (celles partagées par les groupes de recherches) a permis de définir une étape additionnelle de transformation et d'affinement des connaissances des chercheurs. Elle a facilité, entre autres, l'obtention du consensus.

Nous avons ensuite exposé l'environnement conçu pour l'élaboration de la BCons. Cet environnement est représenté par un réseau de cellules de coopération. Nous avons étudié l'architecture d'une cellule où nous trouvons les modules suivants :

1. Une interface graphique qui permet de gérer l'interaction entre le système et l'utilisateur. Elle permet (1) de recevoir les commandes de l'utilisateur et (2) d'accéder aux autres modules de l'environnement.

2. Un module de négociation qui participe à la mise en œuvre du processus de construction de la BCons en (1) favorisant toutes sortes de coopération entre les différents chercheurs impliqués et (2) en contrôlant l'accès concurrent à la base. Les utilisateurs peuvent travailler simultanément sur celle-ci lorsqu'il s'agit d'accéder à des versions différentes. Quand il s'agit de travailler avec la même version de la base, il faut alors suivre le protocole d'accès défini à ce propos.

3. Un module de maintien de la cohérence qui a pour charge de juger toute connaissance soumise à sa propre base de connaissances. Le résultat de ce jugement peut être (1) d'accepter la connaissance dans la base, (2) de refuser la connaissance ou (3) d'accepter la connaissance à titre d'hypothèse. Ce résultat est ensuite transmis au module de versions qui va prendre les mesures nécessaires.

4. Un module de gestion des données internes et un module de transport qui assurent les échanges d'informations entres les différentes cellules de l'environnement.

5. Enfin, un module de gestion de versions qui joue un rôle important dans la construction de la BCons. Ses multiples tâches ont été détaillées dans les protocoles de soumission, de négociation et d'intégration.

   Ce module offre en outre à l'utilisateur la possibilité de sélectionner des instances et des classes appartenant à différentes versions d'une base de collaboration ou de la BCons et de les rassembler au sein de sa base privée. Cette fonctionnalité, appelée configuration, permet à l'utilisateur de définir sa propre vue sur les bases de groupe.

[10] Nous nous sommes limités à définir trois niveaux de bases. Il faut dire que ce découpage peut se réitérer sur un nombre quelconque de niveaux. Ce nombre dépend de la complexité du sujet traité, de sa diversité et du nombre des chercheur y travaillant.

Chapitre 5

Mise en œuvre : le prototype VOG

Nous allons présenter au cours de ce chapitre l'implantation logicielle des différents aspects de notre gestionnaire de versions décrit dans le chapitre 3 et aboutissant au prototype VOG. Deux parties essentielles seront développées : la première décrit la réalisation de l'architecture de versions, la seconde concerne les techniques adoptées afin de mettre en œuvre les opérations de manipulation de versions, y compris le développement d'un langage de requêtes. Les tâches de gestion de versions au sein de l'environnement de construction (Cf. §4.4) n'ont pas été implémentées, l'environnement étant en cours de réalisation.

5.1 Modèle de représentation des versions

Le formalisme choisi pour représenter les différentes versions des bases est le formalisme à objets. L'introduction de notre système au sein de Shirka nécessitant la mise en place de nouvelles entités et de nouveaux schémas, nous avons préféré définir nos versions indépendant de Shirka afin de respecter la généralité de notre modèle. Nous avons implémenté les versions en utilisant le langage de programmation: Le-Lisp. De la sorte, nous n'avons ajouté aucun nouveau schéma aux schémas minimaux du modèle de données.

Nous allons tout d'abord rappeler brièvement les notions introduites précédemment avant de développer leurs implémentation.

5.1.1 Rappel

Nous avons défini au cours du chapitre 3 les entités : *base de connaissances*, *version d'une base* et *axe de travail*. Chacune de ces entités est représentée dans notre prototype par une classe. Les relations de composition existant entre elles sont illustrées figure 5.1. La figure 5.2 rappelle, quant à elle, que chaque version est une liste ordonnée de couches dont le contenu correspond aux connaissances codées par Shirka.

Nous avons distingué au cours du chapitre 4 trois types de bases de connaissances (BC):

- *base Consensuelle*, composée de versions *publiques*;
- *base de collaboration*, composée de versions de *transition*;
- *base de travail*, composée de versions *privées*.

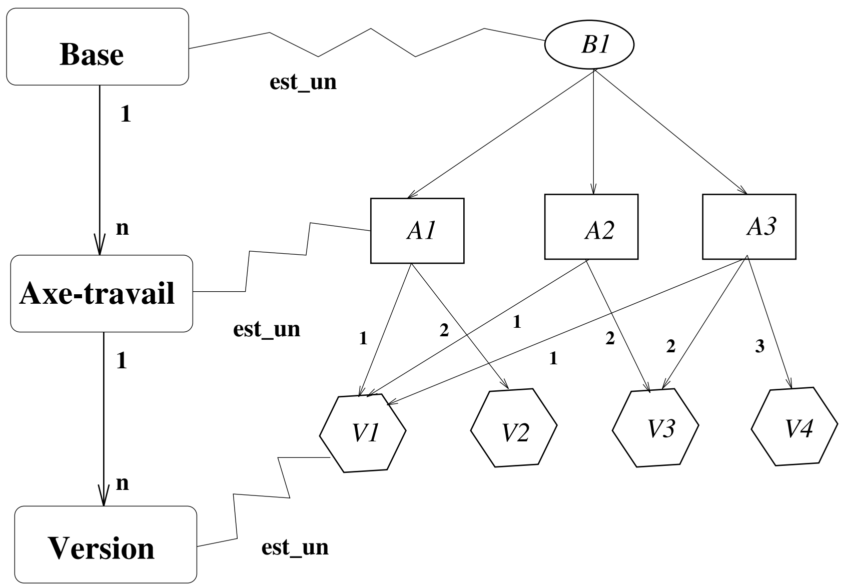


Figure 5.1 - : Une base de connaissances est composée de plusieurs axes de travail, chacun de ces axes étant également composé de plusieurs versions. La relation entre les classes *Base* et *Axe-travail* et également *Axe-travail* et *Version* est donc de type 1 : n.

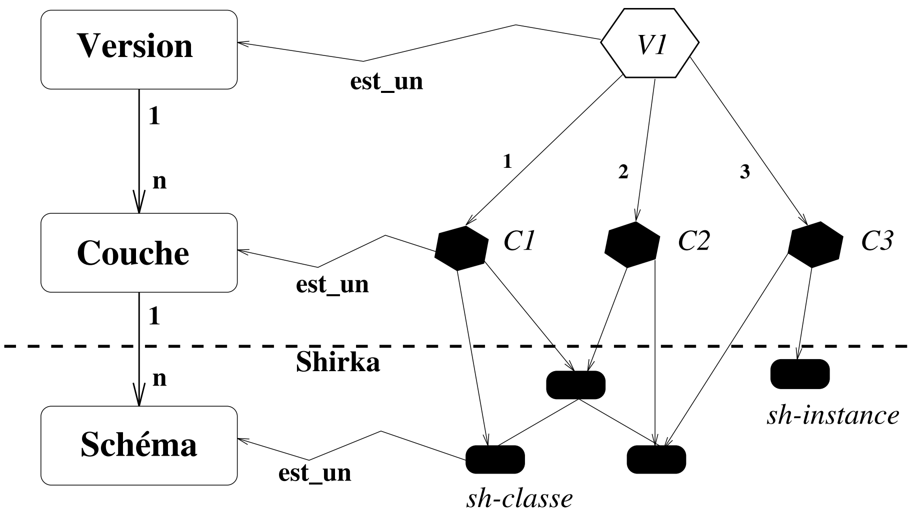


Figure 5.2 - : La relation entre les deux classes *Version* et *Couche* de notre modèle est une relation de composition. La version $V1$ référence dans l'ordre les couches $C1$, $C2$ et $C3$. Il existe également une relation de composition entre la classe *Couche* et la classe *Schéma* de `Shirka`. Par exemple, la couche $C1$ contient les schémas des deux classes, tandis que $C3$ contient le schéma d'une instance ainsi que le schéma de la classe d'appartenance de cette instance.

Chaque BC est liée à une base de données dont le contenu dépend du type de cette première. Toutes ces relations sont rappelées figure 5.3.

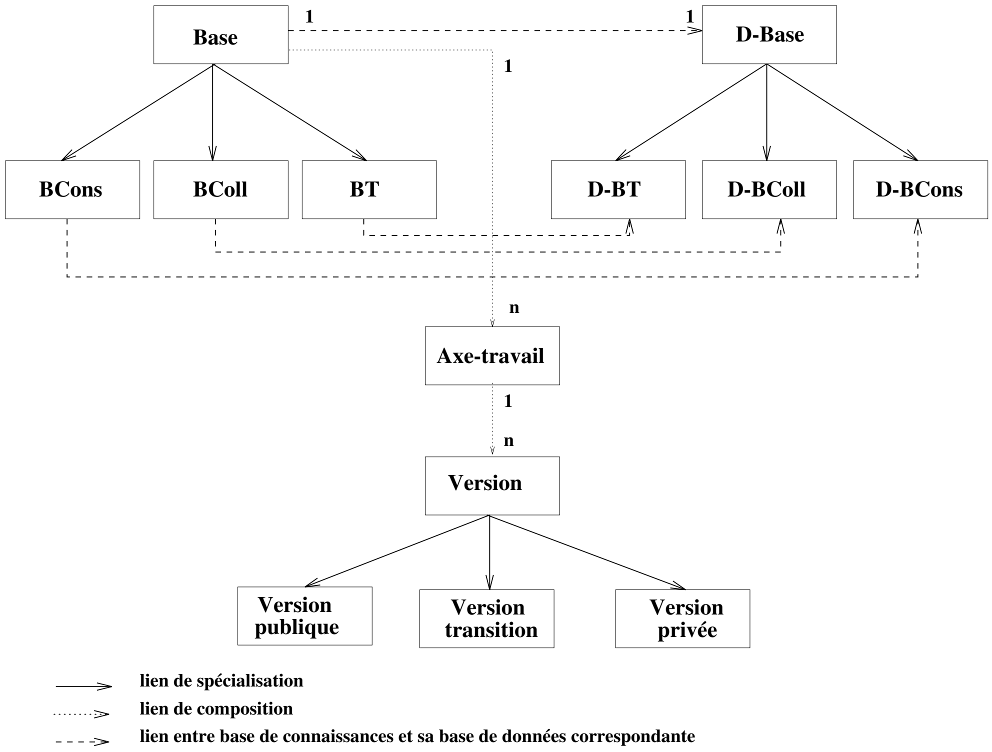


Figure 5.3 - : Cette figure montre trois hiérarchies de classes. La première est celle des bases de connaissances; la seconde est celle des bases de données et enfin la troisième est celle des versions des bases de connaissances.

Ce rappel effectué, nous allons maintenant revenir sur l'ensemble des classes prédéfinies par le gestionnaire de versions afin de modéliser toutes les entités manipulées.

5.1.2 Les classes prédéfinies

Les classes `Le-Lisp` fournies ci-dessous correspondent aux classes que nous avons développées pour notre système de gestion de versions. Chaque classe est décrite par l'ensemble de ses attributs. Le langage `Le-Lisp` permet de définir des types d'objets structurés en utilisant la primitive :

(*DEFSTRUCT* <nom-structure> <champs1> ... <champsn>).

Cette même primitive permet en plus de créer une hiérarchie de types :

(*DEFSTRUCT* <type1>:<type2> <champs1> ... <champsn>)) permet de définir le sous-type <type2> de <type1> avec ses propres attributs (<champs1> ... <champsn>).

La classe ***Base*** est à la racine de la hiérarchie de bases de connaissances. Elle contient une référence à une base de données et une autre à l'ensemble des axes de travail de son arborescence de versions.

(DEFSTRUCT ***Base*** *base-données liste-axes*)

La classe ***D-Base*** est à la racine de la hiérarchie de bases de données. Elle contient toutes les informations détaillées dans la section 4.3.4 et qui sont communes aux trois types de bases de connaissances.

(DEFSTRUCT ***D-Base*** *nom-base nom-reponsable mot-passe no-version-courante date-création-version-courante adresse-base adresse-base-BCons liste-adresses-chercheur*)

Les trois classes ci-dessous sont des sous-classes de ***Base***. L'attribut *base-données* est ainsi spécifié.

(DEFSTRUCT ***Base***:***BT*** *base-données*)

(DEFSTRUCT ***Base***:***BColl*** *base-données*)

(DEFSTRUCT ***Base***:***BCons*** *base-données*)

La classe ***D-BT*** est une sous-classe de ***D-Base***. Son attribut *liste-adresses-BColl* exprime les liens de dépendance entre la BT concernée et les BColl. Il est défini sous forme d'un tableau qui lie le nom d'une BColl et le nom de son projet à l'adresse électronique de son serveur. La valeur de l'attribut hérité *liste-adresses-chercheur* est, dans ce cas, la liste des adresses électroniques de toutes les personnes partageant la (ou les) même BColl que la BT en question.

(DEFSTRUCT ***D-Base***:***D-BT*** *liste-adresses-BColl*)

Les deux classes suivantes sont également des sous-classes de *D-Base*. Elles héritent ainsi tous les attributs de cette dernière. Dans le cas de la classe ***D-BColl***, l'attribut *liste-adresses-chercheur* contient la liste des adresses des chercheurs ayant le droit d'accéder à la BColl correspondante. Cette classe contient de plus l'attribut *nom-projet* de la BColl. Dans le cas de la classe ***D-BCons***, l'attribut *liste-adresses-chercheur* contient les adresses de tous les abonnés à l'environnement de l'élaboration de la BCons. Cette classe possède aussi l'attribut *domaine-recherche* de la BCons.

(DEFSTRUCT ***D-Base***:***D-BColl*** *nom-projet*)

(DEFSTRUCT ***D-Base***:***D-BCons*** *domaine-recherche*)

La classe ***Axe-travail*** contient l'attribut *hypothèse-axe* qui désigne l'hypothèse qui est à l'origine de la création de cet axe. L'attribut *nom-chercheur* indique le chercheur qui est à l'origine de l'axe considéré. L'attribut *axe-chercheur* correspond à la liste des adresses des chercheurs soutenant l'hypothèse de cet axe. Cette classe gère l'ensemble de versions liées à un axe. Elle contient le *no-version-feuille* ayant comme valeur le numéro de la dernière version créée dans cet axe. Cet attribut permet d'une part à savoir le nombre de versions dans l'axe concerné, et d'autre part le numéro de la version courante. La référence *version-feuille* définit le lien entre un axe et l'ensemble de ses versions.

(DEFSTRUCT ***Axe-travail*** *nom-axe hypothèse-axe nom-chercheur version-feuille no-version-feuille axe-chercheur*)

La classe ***Version*** possède l'attribut *date-de-création* dont la valeur sera attribuée par le système lors de la création de la version concernée. L'attribut *nom-version* est le nom donné par l'utilisateur[1]. Chacun de ces deux attributs permet d'identifier une version de manière unique. L'attribut *version-dérivée-de* est une référence à la version à partir de laquelle la version en question est dérivée. C'est par le biais de cette référence que la relation de quasi-héritage est établie. Cette classe possède également l'attribut *couche-correspondante* qui est une référence à la couche correspondant à la version concernée.

(DEFSTRUCT ***Version*** *date-de-création nom-version version-dérivée-de couche-correspondante*)

La *liste de schémas* dans la classe ***Couche*** est un tableau liant le nom d'un schéma (classe, méthode, instance, attribut et facette) à sa représentation interne correspondante au niveau de `Shirka`. Cette donnée est le point d'entrée au modèle de représentation de connaissances. L'attribut *l-schémas-supprimés* contient les noms de tous les schémas supprimés dans la couche associée. Il est complété lors d'une opération de type *sup-instance*, *sup-classe* ou *sup-val-att*. Il faut noter que le schéma lui-même n'est pas détruit physiquement, car il continue à exister pour des autres couches.

(DEFSTRUCT ***Couche*** *l-schémas l-schémas-supprimés*)

5.1.3 Représentation interne des schémas dans Shirka

Un schéma de n'importe quelle entité de `Shirka` est représenté en interne par un vecteur `Le-Lisp` que nous allons maintenant détailler afin de faciliter la compréhension de la section suivante. Nous allons ainsi donner cette structure interne pour les schémas de classes, de méthodes, d'instances, d'attributs et de facettes.

- La représentation interne d'une classe et d'une méthode contient les informations suivantes :
  <est-un> <nom-sch> <liste-att> <sorte-de> <spec> <inst> où

  - *est-un* est le nom de la classe d'appartenance d'une classe, qui est la classe **Schéma**. En effet, `Shirka` utilise le principe de méta-schéma permettant de concevoir un schéma de classe comme une instance de la méta-classe **Schéma**;
  - *nom-sch* est le nom du schéma de classe;
  - *liste-att* est la liste des attributs propres (et non pas hérités) de la classe. Chacun de ses attributs est représenté par son vecteur interne;
  - *sorte-de* est le nom de la sur-classe directe de la classe;
  - *spec* est la liste des sous-classes directes de la classe;
  - *inst* est la liste des propres instances de la classe.

- La représentation interne d'une instance est donnée par :
  <est-un> <nom-inst> <val-att1> <val-att2> …<val-att*n*> où

  - *est-un* est le nom de la classe à laquelle l'instance est liée;

[1] La valeur de cet attribut est facultative

  – *nom-inst* est le nom de l'instance;

  – *val-att*1 ... *val-attn* correspond aux valeurs des attributs de l'instance.

- La représentation interne pour un attribut est donnée par :
  <est-un> <nom-sch> <nom-att> <liste-fact> <nature> <type> <dom> <inter> <sauf> où :

  – *est-un* a comme valeur *attribut*. En effet, chaque attribut est une instance de la classe **Attribut** de `Shirka`;

  – *nom-sch* contient le nom du schéma de classe de l'attribut concerné;

  – *nom-att* est le nom de l'attribut;

  – *liste-fact* est la liste des facettes de description de l'attribut;

  – *nature* correspond à la nature de l'attribut (mono ou multi-valué);

  – *dom* est la liste des valeurs énumérées du domaine de l'attribut;

  – *inter* est l'intervalle des valeurs de l'attribut;

  – *sauf* correspond aux valeurs interdites pour l'attribut.

- La représentation interne d'une facette contient les informations :
  <nom-fact> <est-un> <list-val> où :

  – *nom-fact* est le nom de la facette;

  – *est-un* est le nom de la classe de la facette;

  – *list-val* est le type des valeurs de la facette.

Lors de la création d'une classe ou une instance, leurs vecteurs respectifs sont intégrés dans la couche courante. Par exemple, la classe **Etudiant** dont la description externe est donnée figure 5.4, est remplacée dans les couches par le vecteur :
#[schema Etudiant (#[attribut () âge (#[$intervalle () (#[intervalle () 7 30])]) $un entier () ((7 30)) ()] #[attribut () no-carte (#[$un () entier]) $un entier () ()]) (Personne) (Etudiant-diplômé) (E0)]
Notons que parmi la liste de ses attributs ne se trouvent que ceux locaux à cette classe. L'instance *E0* de la figure 5.4 est quant à elle remplacée par le vecteur :
#[Etudiant E0 "martin" 20 D0 123455]
Il faut remarquer que l'instance *E0* possède les valeurs des attributs hérités de la classe **Etudiant** et les valeurs des attributs de sa classe. L'ordre de ces valeurs dans le vecteur `Le-Lisp` est important, puisque la première valeur correspond au premier attribut et ainsi de suite.

Il est finalement indispensable de répéter que la représentation interne des connaissances d'une base ainsi que les structures des classes du système de version sont cachées à l'utilisateur. Les seules entités directement manipulables sont les bases et leurs versions. L'ensemble des opérations de manipulation (i.e. interface fonctionnelle) fournies par le gestionnaire est décrit dans §**??**.

Nous allons maintenant détailler au cours de la section suivante la réalisation de cette interface fonctionnelle.

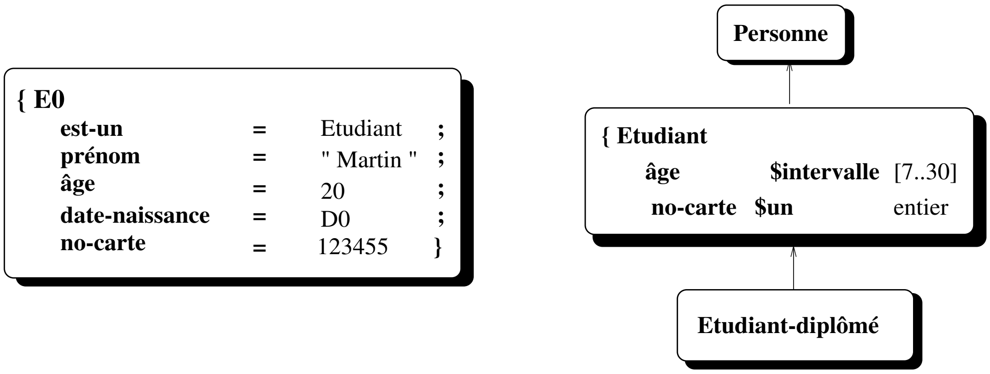


Figure 5.4 - : La hiérarchie de la classe **Etudiant** (à droite) et le schéma de l'instance *E0* (à gauche).

5.2 Interface fonctionnelle du gestionnaire de versions

Cette interface contient deux modules : le premier (module 1 dans figure 5.5) concerne les opérations de manipulation des bases de connaissances et leurs versions. Ce module a été développé en langage `Le-Lisp` au-dessus de `Shirka`. Le second (module 2 dans figure 5.5) concerne la mise en œuvre des opérations de manipulation des versions de schémas de connaissances dans `Shirka`.

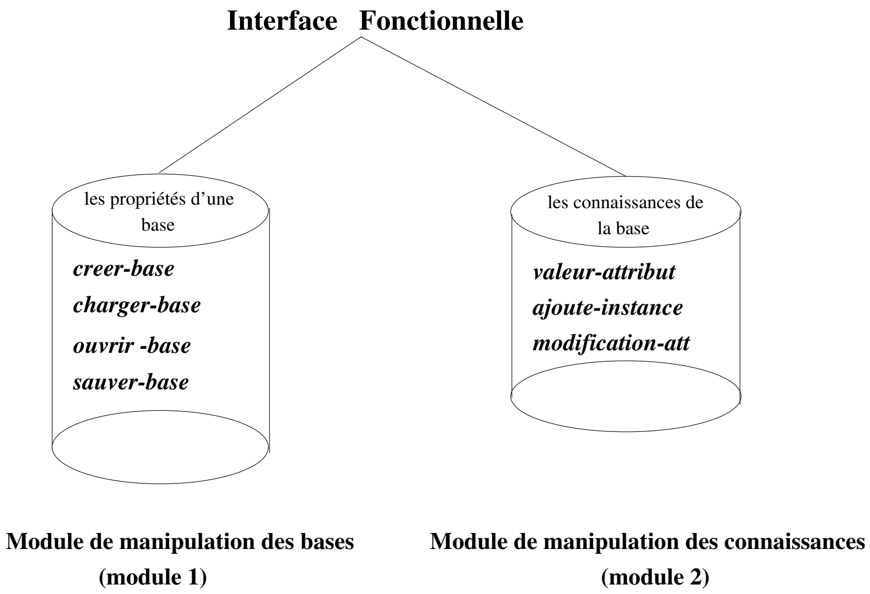


Figure 5.5 - : Les deux modules de l'interface fonctionnelle du prototype `VOG`.

L'ensemble des opérations décrites ci-dessous se déroule sous l'environnement `Le-Lisp natif (V15.25)` [Ilog, 1991]. Le prototype `VOG` est installé sur des stations de travail SUN.

L'utilisation du gestionnaire de versions requière au préalable le chargement des fichiers sources `VOG` et `Shirka` correspondant aux deux systèmes respectifs. Ce chargement peut être réalisé au travers de l'interface graphique de la cellule de coopération.

5.2.1 Réalisation des opérations de manipulation des bases et de leurs versions

Il s'agit des opérations de manipulation des deux classes **Base** et **Version**, ou plus précisément des instances de ces deux classes. L'accès à ces instances n'est possible que via les opérations de l'interface fonctionnelle. La modification par exemple du nom d'une version ne pourra pas être effectuée en changeant directement la valeur de l'attribut *nom-version* de la version correspondante. Il faudra obligatoirement utiliser la fonction prévue à cet effet. L'intérêt essentiel d'une telle approche est que tous les détails de l'implémentation de représentation, de gestion et du stockage des versions et de leurs bases sont cachés à l'utilisateur final. Celui-ci ne tient compte que des services offerts.

Cette vue fonctionnelle d'une BC a été introduite pour la première fois par Brachman pour son système de représentation *Krypton* [Brachman et al., 1983]. L'interface proposée ne comporte par contre que les deux opérations *ASK* et *TELL* ne permettant pas en particulier d'obtenir une explication sur les résultats obtenus lors d'une consultation. Il faut également noter que les deux seules réponses possibles lors de ces consultations sont uniquement *oui* et *non*. On retrouve également ce type d'approche fonctionnelle dans les système Krisys [Mattos, 1988] et Telos [Mylopoulos et al., 1990].

Nous allons maintenant revenir plus en détail sur l'implémentation des différentes commandes disponibles. Les traces d'exécution de ces commandes sont données au cours de l'annexe A.

Créer une base

La création d'une base est réalisée grâce à la commande *creer-base* possédant les trois paramètres *<nom-base>*, *<nom-projet>*, *<domaine-de-la-base>*. L'exécution de cette commande dépend de l'endroit où elle a été appelée :

1. cette commande a été lancée à partir du serveur de la BCons de l'environnement. Il faut alors que l'administrateur de celle-ci (qui doit être la même personne effectuant la commande) fournisse le nom de la base et son domaine. Un objet de la classe **BCons** est créé ainsi qu'un objet de la classe **D-BCons**. Les attributs du dernier objet auront comme valeurs les informations fournies. Le système complétera automatiquement les autres attributs : *date-de-création*, *nom-reponsable* et *adresse-serveur-BCons*.

2. la commande a été lancée depuis un serveur d'une BColl déclarée dans l'environnement. Le nom de la base et le nom du projet de recherche sont tous les deux exigés puisqu'ils servent à identifier la base créée. Comme précédemment, les deux classes **BColl** et **D-BColl** sont instanciées et leurs attributs reçoivent les valeurs adéquates.

3. La commande a été exécutée à partir d'une machine autre que les deux mentionnées ci-dessus. Le nom de la base suffit alors pour la créer (i.e. créer une instance de la classe **BT**). Le nom fourni est inséré dans l'instance de la classe **D-BT** aussi créée.

La base étant créée, la seconde étape consiste ensuite à y introduire les différentes connaissances. Les bases publiques sont remplies automatiquement par soumission de connaissances. Les bases privées, quant à elles, sont élaborées par leurs chercheurs. Ceux-ci ont le choix entre en deux commandes *remplir-base* et *configurer-base*.

Remplir une base

L'utilisateur choisit cette commande pour initialiser sa base en incorporant ses propres connaissances, ou en réutilisant une version déjà existante (Cf. §4.4.1). Cette commande accepte un des deux arguments suivants (le choix de l'utilisateur est fait en donnant une valeur à un des deux arguments) :

- <*fichier*> : il s'agit du nom de fichier contenant les descriptions des schémas de classes, d'instances et de méthodes. Dans ce cas, une instance de la classe **Axe-travail** est créée et les noms de l'axe et de son hypothèse sont demandés. Au sein de cette instance, l'attribut *version-feuille* n'est qu'une instance de la classe **Version**. Cette version contient la première couche (instance de la classe **Couche**) référençant les schémas fournis par le fichier. Chaque nom de schéma sera associé à son vecteur interne (voir fig. 5.6).

{ Base1
base-données D-base1 ;
liste-axe [Axe1] }

{ Axe1
nom-axe "axe-courant";
nom-chercheur "ntayar";
version-feuille Version1;
no-version-feuille 1 }

{ Version1
date-de-création "08/03/95";
couche-correspondante [couche1]
}

{ D-base1
nom-base "Coligène";
nom-reponsable "ntayar";
no-version-courante 1;
adresse-serveur-BCons "cosmos@imag.fr"
}

{ Couche1
liste-schémas

Personne	[shema, nom, âge,.. (p1).]
Etudiant	[schema, âge, no-carte,..]
p1	[Personne, "martin",...]

}

Figure 5.6 - : Un exemple d'instances de base de connaissances, d'axe de travail et de version.

- <*version*> : il s'agit de l'identificateur d'une version privée, de transition ou publique. Rappelons que pour les trois types, une version est identifiée de manière unique par le nom de sa base de connaissances, le nom de son axe d'appartenance et son numéro (ou la date de création) dans cet axe. Ces trois données doivent donc obligatoirement être fournies. Au moyen de cet identificateur, toute autre base pourra référencer sur la même machine une version qui ne lui appartient pas (voir fig. 5.7).

  Enfin, dans le cas où la base de la version demandée ne se trouve pas dans la même machine que la base créée, le système va charger la version voulue. Celle-ci se voit attribuer un nouvel identificateur : le triplet (nouvelle base d'appartenance, nouveau axe de travail, le numéro dans cet axe) sur le site. Dans tous les cas, la version identifiée forme la première version de la base à remplir. Une deuxième version sera ensuite créée afin de recevoir les modifications de l'utilisateur. Les deux

```
{ Base2                                  { D-base2
   base-données    D-base2 ;                nom-base                  "Société";
   liste-axe       [Axe1]      }            nom-reponsable            "ntayar";
                                            no-version-courante       1;
{ Axe1                                      adresse-serveur-BCons     "cosmos@imag.fr"
   nom-axe         "axe-1";                                     }
   nom-chercheur   "ntayar";
   version-feuille ("Française", Axe2, 2);
   no-version-feuille 1        }
```

Figure 5.7 - : Exemple d'une base *société* référençant une version qui ne lui appartient pas. La première version de *Axe1* de cette base n'est que la deuxième version de l'axe *Axe2* de la base *Française*.

versions appartiendront au même axe (i.e. instance de la classe **Axe-travail**) dont le nom est réclamé par le système.

Modifier une base

Lorsqu'un utilisateur veut effectuer un changement sur une base (de n'importe quel type), il doit tout d'abord la charger dans son espace local (la commande est *charger* <*nom-base*>). S'il est autorisé à le faire, la base chargée devient alors une base de travail et la totalité de ses versions est accessible.

Étant donné que nous avons permis à l'utilisateur de charger le nombre de bases qu'il désire, nous lui avons fourni la commande *ouvrir* <*nom-base*> permettant de spécifier la base sur laquelle il veut travailler. Cette commande signale le début d'une session de consultation de n'importe quel axe ou version de la base à condition que ces deux informations soient précisées par l'utilisateur lors de chaque commande. Si ce dernier ne les fournit pas, le contexte de travail courant est alors pris en compte. Celui-ci est composé de la dernière version créée et de son axe d'appartenance.

Pour apporter des modifications, il faut lancer la commande *mise-a-jour* acceptant comme paramètres le numéro de la version qui va être modifiée ainsi que son axe d'appartenance. À partir de ce moment, une nouvelle version est dérivée en créant automatiquement sa couche correspondante (i.e. les deux classes **Version** et **Couche** sont instanciées). Celle-ci recevra ainsi toutes les modifications.

Sauver une base

Pour sortir de cette session, la commande *sauver-base* permet à la fois d'enregistrer toutes les opérations effectuées, de déclarer la nouvelle version créée comme "figée" et de retourner à la session de consultation. Si l'utilisateur veut quitter son application sans sauvegarder, il tape la commande *fermer-base*. Dans ce cas, la nouvelle version créée (ainsi que sa couche) est supprimée et son contenu est perdu. Un message d'avertissement est affiché afin d'annoncer les conséquences d'une telle commande. *fermer-base* peut également être utilisée afin de sortir de la session de consultation de la base courante. L'utilisateur est ensuite libre d'ouvrir une autre base chargée ou de quitter l'environnement de travail.

Afin d'éviter le stockage d'un important volume de données, nous avons choisi le *stockage différentiel*. Cette technique consiste à stocker la différence entre deux versions successives d'une base. Par conséquent le système ne sauvegarde pour chaque axe de travail que les couches créées dans cet axe. Afin de reconstituer les autres versions de l'axe, nous avons le choix entre garder la version finale et lui soustraire des couches ou garder la versions initiale et lui ajouter des couches. Nous avons préféré la première solution car en général, ce sont les versions les plus récentes qui sont les plus consultées ou mises à jour.

La figure 5.8 illustre l'enchaînement de l'ensemble de ces commandes.

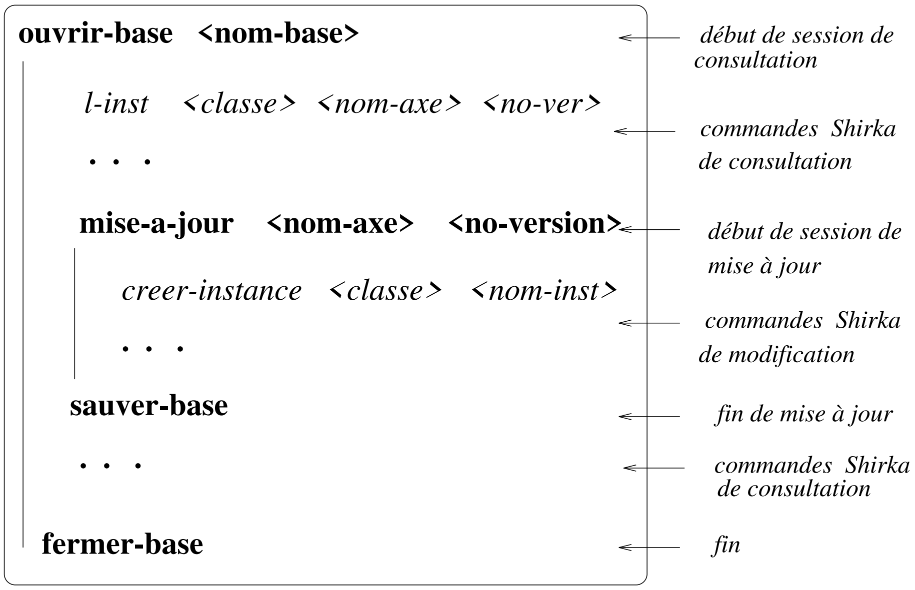


Figure 5.8 - : Ordre des commandes de consultation et de modification d'une base.

5.2.2 Réalisation des opérations de manipulation des versions de classes et d'instances

Nous allons nous intéresser au cours de cette section aux opérations de consultation et de mise à jour, la mise à jour pouvant être une modification, une addition ou une suppression d'un schéma de classe ou d'instance. Les traces d'exécution de ces opérations sont données au cours de l'annexe A.

Opérations de consultation

Dans le cadre des opérations de consultation, nous avons réécrit l'ensemble des commandes *Shirka* afin de prendre en compte les deux nouveaux paramètres qui sont le nom de l'axe de travail et l'identificateur de version (que ce soit le numéro, la date de création ou un instant T). La liste des commandes de consultation de `Shirka` est donc maintenant la suivante :

- *l-att* <*classe*> <*axe*> <*version*> permettant d'obtenir la liste des attributs d'une version d'une classe appartenant à un axe de travail donné;
- *l-inst* <*classe*> <*axe*> <*version*> fournissant la liste des instances d'une version d'une classe appartenant à un axe de travail donné;

- *l-spec* <*classe*> <*axe*> <*version*> permettant d'obtenir la liste des sous-classes directes et indirectes d'une version d'une classe appartenant à un axe de travail donné;

- *vi* <*instance*> <*axe*> <*version*> permettant de visualiser à l'écran une version d'une instance (affichage des noms et des valeurs de chaque attribut);

- *val?* <*instance*> <*attribut*> <*axe*> <*version*> permettant de consulter la valeur d'un attribut dans une version d'une instance.

Opérations de mise à jour

Dans le cadre des opérations de mise à jour, nous avons implémenté l'ensemble des fonctions de gestion de la cohérence exposées section 3.5. Afin d'économiser l'espace mémoire, nous avons choisi de ne stocker au niveau de chaque couche que les informations ayant été modifiées par rapport à la couche précédente.

Considérons par exemple l'instance *E0* de la classe **Etudiant** définie par les valeurs suivantes : prénom = "Martin", âge = 20, adresse = D0. Le vecteur interne de *E0* est :
#[Etudiant E0 "Martin" 20 D0]
Pour chacune des opérations suivantes, le système ne recopie pas complètement le vecteur précédent, il le remplace par un vecteur plus petit, approprié à l'opération effectuée :

- modification de la valeur d'un attribut. Par exemple la valeur de l'attribut âge est remplacée par 25. Le vecteur inséré pour représenter ce changement est : #[E0 (âge 25)].

- addition d'une valeur d'un attribut. Par exemple l'utilisateur donne la valeur 123455 à l'attribut *no-carte* de l'instance *E0*. Le vecteur inséré pour représenter cette opération est : #[E0 (no-carte 123455)].

- suppression de la valeur d'un attribut. Par exemple la valeur de l'attribut âge est supprimée. Cette opération peut être vue comme un cas particulier de modification d'une valeur d'un attribut. Cette opération est représentée au niveau de la couche par le vecteur : #[E0 (âge)].

De même, dans le cas de la classe **Etudiant**, la définition d'une nouvelle sous-classe **Thésard** ajoutée à sa liste de spécialisation entraînera la réinsertion de son vecteur réduit

#[Etudiant (l-spec Etudiant-diplômé Thésard)] au lieu de son vecteur complet

#[schema Etudiant (#[attribut () âge (#[$intervalle () (#[intervalle () 20 30])]) $un entier () ((20 30)) ()] #[attribut () no-carte (#[$un () entier]) $un entier () ()]) (personne) (Etudiant-diplômé Thésard) (E0)].

L'approche que nous avons suivie présente comme avantage de ne pas surcharger nos couches mais exige en contre-partie l'écriture de fonctions supplémentaires permettant de reconstituer le schéma complet d'une connaissance à partir des morceaux de vecteurs retrouvés au sein des différentes couches. Nous avons ainsi dû remplacer la fonction **schéma** de `Shirka` par notre fonction **construire-schéma**. L'algorithme est décrit dans le tableau 5.1.

L'algorithme présenté tableau 5.2 décrit la fonction **chercher-val** qui est la fonction principale de l'algorithme de **construire-schéma**.

Algorithme 1 *construire-schéma(S, A, No-V)*

Entrées : un nom de schéma S, un nom d'axe de travail A et un numéro de version No-V.
Sortie : la représentation interne du schéma S par rapport à la version V.
Remarque : Le schéma S, s'il existe, est représenté dans la version concernée par un vecteur de la forme :

- pour une classe : [S (sorte-de val-sorte-de) (l-att val-l-att) (l-inst val-l-inst) (l-spec val-l-spec)]
- pour une instance : [S (est-un val-est-un) (att1 val1) ... (attn valn)]

Dans une couche C, S peut avoir un seul "morceau" de son vecteur (par exemple [s (l-att val-l-att))]. Le but est de retrouver le vecteur complet de S à travers toutes les couches à partir de celle correspondante à la version *No-V*.

V ← **trouver-version** *(A , No-V) /* la fonction* **trouver-version** *rend une référence à la version dont le numéro est* No-V *et le nom de son axe d'appartenance est* A **/*

Si S *est un nom de classe* **alors**
  vecteur-valeur ← **chercher-val** *([*l-att, sorte-de, l-spec, l-inst*],* S, V*)*
  vecteur-interne ← [schéma, (), S, vecteur-valeur[1], vecteur-valeur[2], vecteur-valeur[3], vecteur-val[4]]
Fin Si
Si S *est un nom d'instance* **alors**
  list-att ← **chercher-val** *([*l-att*],* classe-de-l'instance, V*)*
  vecteur-valeur ← **chercher-val** *([*est-un, list-att*],* S, V*)*
  vecteur-interne ← [vecteur-valeur[1], (), S, vecteur-valeur[2], ..., vecteur-valeur[n]]
Fin Si
Rendre *vecteur-interne*

Tableau 5.1 - : L'algorithme de construction d'une version d'un schéma de connaissance à partir de ses "morceaux" éparpillés dans les versions précédentes de la base.

Algorithme 2 *chercher-val ([info1, ..., infon], S, V)*

Entrées : [info1, ..., info$n$] est le vecteur d'information à chercher, S est le schéma, V est la version concernée.
Sortie : vecteur-val-info qui est le vecteur des valeurs de toutes les informations info$i$ dans S par rapport à V.

vecteur-info ← *[info1, ..., info$n$]*
L ← *longueur (vecteur-info)*
vecteur-val-info ← *[()]*
Si $S \in$ V.couche-correspondante **alors**
/ i.e. que S possède un vecteur-S dans la couche correspondante à V */*
  Pour *chaque info* ∈ *vecteur-info* **faire**
    chercher si info appartient au vecteur-S.
    Si *oui* **alors**
      / ajouter la valeur trouvée* val-info *au vecteur de valeurs */*
      vecteur-val-info ← *vecteur-val-info* + *val-info*
      / enlever info du vecteur car sa valeur a été trouvée */*
      vecteur-info ← *vecteur-info - info*
    Fin Si
  Fin Pour
Fin Si
Sinon
/ S n'appartient pas à la couche de V */*
  / il faut chercher si S n'a pas été supprimé au niveau de la couche concernée */*
  Si $S \in$ V.couche-correspondante.l-schéma-supprimés **alors**
  / S a été supprimé dans la couche de V */*
    Écrire *(Valeur non trouvée car le schéma S n'existe plus dans la version V)*
  Fin Si
  Sinon
  / descendre un pas dans la hiérarchie de versions et chercher dans la couche inférieure de la couche de V */*
  V ← *V.mère*
  / appeler de nouveau la fonction* **chercher-val** *avec la nouvelle version V */*
  chercher-val *([info1, ..., info$n$], S, V)*
Fin Sinon
Rendre *vecteur-val-info*

Tableau 5.2 - : L'algorithme de recherche des valeurs d'un schéma de connaissance.

En résumer, les deux algorithmes 5.1 et 5.2 ne sont que l'implémentation de la relation de *quasi-héritage* définie au §3.3.2; le principe dans les deux cas est le même : parcourir les couches à partir de la couche correspondant à la version donnée afin de trouver les valeurs désirées. Si une couche ne contient pas l'information recherchée, il faut alors parcourir sa couche inférieure ainsi de suite jusqu'à la première couche. La complexité des deux algorithmes est O(n), où n est le nombre de couche à parcourir. C'est donc une complexité linéaire avec le nombre de couches.

5.3 Langage de requêtes

Nous allons nous intéresser au cours de cette dernière section aux requêtes qu'un chercheur de l'environnement peut adresser aux différentes bases de connaissances. Nous préciserons tout d'abord le langage de requêtes utilisé. Nous nous intéresserons ensuite, au travers d'exemples, à la traduction d'une requête par le gestionnaire de versions en terme d'opérations de sélection.

Le système actuel `Shirka` permet d'interroger les valeurs d'attributs de l'ensemble des classes définies dans une base. Sous l'interface d'interrogation IVAN [Grivaud, 1992], il est également possible d'utiliser les opérateurs booléens *ET*, *OU* et *SAUF* afin de joindre plusieurs requêtes posées sur une classe particulière. Notre travail a consisté à adapter ce même langage afin de sélectionner non plus des instances mais des versions d'instances. Nous avons eu besoin de l'étendre afin d'incorporer la nouvelle dimension qui est le temps. Nous avons ainsi rajouté les deux arguments *nom-axe* et *identificateur-version* pour chaque requête posée.

En plus des extensions apportées aux requêtes existantes, nous avons enrichi le langage de requêtes afin de :
(1) fournir quelques opérateurs temporels et (2) avoir la possibilité d'interroger non seulement les connaissances dans la base mais de poser des questions sur les bases elles-mêmes. Ces deux aspects particuliers vont être développés dans les deux prochaines sous-sections.

Tous les exemples fournis ci-après sont donnés en langue naturelle pour la clarté du sens. Il faut signaler que le système ne traite pas ces questions telles qu'elles sont posées. En fait, les mots-clés réservés au langage de requêtes ainsi que les opérateurs de ce dernier sont affichés dans l'interface graphique. C'est à l'utilisateur de former ensuite ses propres interrogations à partir de ce vocabulaire réduit.

5.3.1 Opérateurs temporels

Il s'agit de formuler des requêtes d'interrogation utilisant des opérateurs temporels[2]. Dans la littérature associée aux bases historiques, Tuzhilin [Tuzhilin et Clifford, 1990] et Gadia [Gadia et Yeung, 1988] définissent les opérateurs temporels les plus importants. Ces opérateurs sont inclus dans la liste de ceux que nous avons proposés et qui est fournie ci-dessous. Quelques-uns d'entre eux ont déjà été cités au cours de la section 3.4.2.

Les opérateurs de notre modèle prennent comme arguments les noms de la base et de son axe de travail où la recherche doit être effectuée. Prenons par exemple le cas de l'axe de travail (voir fig. 5.9). Nous supposons ainsi que toutes les requêtes suivantes ont été posées sur les connaissances dans cet axe. Nous allons citer les opérateurs temporels supportés

[2] Comme nous l'avons déjà cité au cours de la section 2.4, nous ne réalisons dans notre système qu'une gestion élémentaire du temps.

par notre système. Pour chaque opérateur, nous précisons la syntaxe et la sémantique accordées par notre modèle.

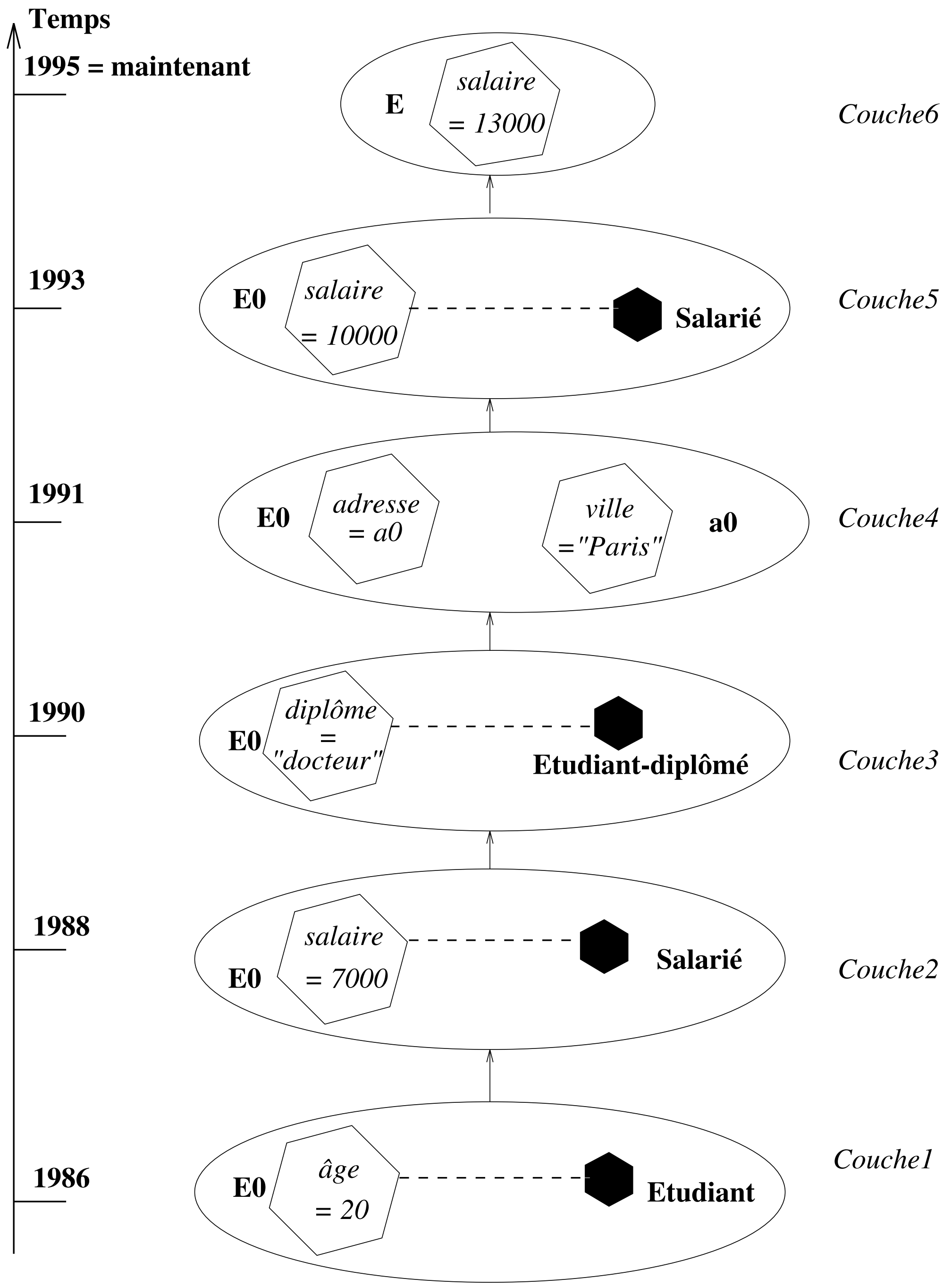


Figure 5.9 - : Exemple d'un axe de travail qui montre l'historique de l'instance E de 1986 jusqu'à maintenant.

Depuis-quand **<condition>**[3] extrait le moment le plus récent pour lequel la *condition* est devenue vraie. Si la condition est fausse au temps présent, la réponse renvoyée est un message d'erreur. Par exemple :

Depuis-quand *E0* vit à *Paris*?

L'utilisateur doit taper la requête : ***depuis-quand*** [**E0.adresse.(ville="Paris")**].

La réponse est 1991.

Quand <**condition**> extrait les intervalles de temps pendant lesquels la *condition donnée* est vérifiée. Par exemple :
Quand *E0* était *étudiant*?
L'utilisateur doit taper : ***quand*** [**E0.(est-un=Etudiant)**].
La réponse est les intervalles *[1986 1988[* et *[1990 1993[*.

Depuis <**temps**> <**condition**> extrait les entités vérifiant la *condition* donnée à partir du *temps* déterminé jusqu'à maintenant. Par exemple :
Depuis 1988 quels sont les *employés* ayant un *salaire* $\geq$ 10000?
L'utilisateur doit taper : ***depuis*** 1988 [**X.(est-un=Salarié)** et **X.(salaire $\geq$ 10000)**].
La réponse est X = *E*.

Jusqu'à <**temps**> <**condition**> extrait les entités vérifiant la *condition* donnée à partir de la date de création de l'axe concerné jusqu'à *temps*. Par exemple :
Jusqu'à 1995 quels sont les *salaires* de l'employé *E0*?
L'utilisateur doit taper : ***jusqu'à*** 1995 [**E0.salaire** et **E0.(est-un=Salarié)**]
La réponse est salaire = 7000, 10000, 13000.

Entre <**temps1**> <**temps2**> <**condition**> extrait les entités vérifiant la *condition* donnée durant l'intervalle du temps [*temps1 temps2*]. Par exemple :
Entre 1986 et 1994 quel était le *statut* de *E0*?
L'utilisateur doit taper : ***entre*** 1986 et 1994 [**E0.est-un**].
Pour cette requête, il faut chercher les classes d'appartenance de *E0* pendant la période donnée. Ainsi, la réponse est *Etudiant*, *Salarié*, *Etudiant-diplômé*, *Salarié*.

Plus-longue-période <**condition**> extrait la plus longue période durant laquelle la *condition* est vérifiée. Le système va chercher tous les intervalles de temps pour lesquels la condition est vérifiée (comme pour l'opérateur ***Quand***). Il choisira ensuite le plus grand intervalle. Par exemple :
Plus-longue-période où *E0* était *Etudiant*?
L'utilisateur doit taper : ***plus-longue-période*** [**E0.(est-un=Etudiant)**]
La réponse est *[1990 1993[*.

Plus-courte-période <**condition**> extrait la plus courte période durant laquelle la *condition* est vérifiée. La façon de procéder est similaire à celle de l'opérateur précédent.

Quelques opérateurs temporels peuvent être exprimés par l'intermédiaire des autres. Par exemple : l'opérateur *entre* $t1$ et $t2$ est équivalent aux deux opérateurs *depuis* $t1$, *jusqu'à* $t2$ utilisés ensemble. Un autre exemple est celui de l'opérateur *jusqu'à* t où t désigne le moment présent. Dans ce cas d'utilisation particulière, cet opérateur peut être remplacé par *quand*. Prenons l'exemple de la requête :
jusqu'à 1995 quels sont les *salaires* de l'employé *E0*?. Cette requête peut être remplacée par :
quel était le *salaire* de *E0* ***quand*** il était *Salarié*?
Pour les deux requêtes, le système va rechercher les périodes dans lesquelles *E0* était *salarié*. Il va ensuite extraire, pour chacune de ces périodes, la valeur de l'attribut *salaire*. La réponse est 7000, 10000, 13000.

5.3.2 Requêtes sur les bases de connaissances et leurs versions

Les bases de connaissances ainsi que leurs versions étant modélisées par des classes, il est possible pour les utilisateurs de notre système de poser leurs requêtes sur les attributs associés à ces classes. Ils peuvent ainsi se renseigner sur la date de création d'une base, sur le nom du reponsable d'une base de collaboration ou sur l'adresse électronique d'un des clients de l'environnement. Voici quelques exemples de questions pouvant être posées :

- *Quel est le nom du chercheur qui est à l'origine de l'existence de l'axe* A1 *de la base de collaboration dont le nom du projet est* Sherpa?
  L'utilisateur doit taper :
  [***BColl.base-données.(nom-projet="Sherpa") et BColl.(A1 ∈ liste-axes)***] et [**A1.nom-chercheur**].

- *Quel est la description consensuelle de l'entité* génome *dans le domaine de la biologie moléculaire*?
  L'utilisateur doit taper :
  [***BCons.base-données.(domaine-recherche="biologie moléculaire") et BCons.base-données.date-création-version-courante.génome.l-att***].
  Le système va donc choisir la BCons dont le domaine de recherche est la biologie moléculaire. Pour cette base, il va extraite la date de création de sa version courante. Il va ensuite trouver la liste des attributs de la classe *génome* par rapport à cette date.

- *Quelle est la date de mise à jour de la base consensuelle en médecine*?
  L'utilisateur doit taper :
  [***BCons.base-données.(domaine-recherche="médecine") et BCons.base-données.date-création-version-courante***].
  La date de création de la version courante de la BCons correspond à la date de sa mise à jour.

- *Quels sont les adresses des chercheurs supportant l'hypothèse de l'axe* A2 *de la base de collaboration* B1?
  L'utilisateur doit taper :
  [***B1.(A2 ∈ liste-axes)***] et [**A2.axe-chercheur**].
  L'attribut axe-chercheur de la classe Axe-travail correspond à la liste des adresses des chercheurs qui soutiennent l'axe concerné.

Pour chacune de ces requêtes, le gestionnaire de versions va préparer le message adéquat à envoyer à la cellule destinataire (dont l'adresse se trouve dans la base de données de la cellule source). Le système de celle-ci va tout d'abord sélectionner les classes concernées et ensuite exécuter les recherches nécessaires des instances ou des valeurs d'attributs afin de répondre. Nous allons revenir plus en détail sur ce mécanisme au cours de la section suivante.

5.3.3 Exécution des requêtes

Nous allons aborder au cours de cette section l'exécution d'une requête par le VOG. L'algorithme est donné au travers d'exemples significatifs.

Requêtes historiques

Prenons l'exemple de la requête historique :
Entre 1986 et 1994 quel était le statut *de* E0 ?.

Nous considérerons que cette requête est posée pour l'axe de travail *(AT)*. Le système exécute la fonction **requête-historique** (est-un, E0, AT, 1989, 1994) dont l'algorithme[4] est donné tableau 5.3.

Algorithme 3 ***requête-historique*** *(info, S, AT, t1, t2)*

Entrées : info est l'information recherchée, S est le schéma, AT est le nom de l'axe concerné, t1 est la date de départ et t2 et la date d'arrivée.
Sortie : la liste des valeurs de info entre t1 et t2 par rapport à l'axe AT.

réponse $\leftarrow$ *()*
/ De la liste des versions de l'axe AT, le système détermine la première version dont la date de création* $\geq$ *t1 (c-à-d V-t1) et la dernière version dont la date de création* $\leq$ *t2 (c-à-d V-t2) */*
Répéter
    / appeler la fonction chercher-val avec la version V-t2 */*
    réponse $\leftarrow$ *réponse +* **chercher-val** *(info, S, V-t2)*
    / la version V-t2 est remplacée par la version qui précède la version d'arrêt de la fonction chercher-val. Cette dernière doit alors être appelée avec la nouvelle valeur de V-t2 */*
    V-t2 $\leftarrow$ *version qui précède la version d'arrêt de chercher-val*

Jusqu'à ce que *V-t2 = V-t1.version-dérivée-de*
Rendre *réponse*

Tableau 5.3 - : L'algorithme de réponse aux requêtes historiques.

L'exécution de **requête-historique** (est-un, E0, AT, 1989, 1994) (pour répondre à la requête donnée ci-dessus) se déroule de la manière suivante :

1. V-t2 $\leftarrow V5$ et V-t1 $\leftarrow V2$ (voir fig. 5.9).
2. **chercher-val**(est-un, E0, V5).
3. réponse $\leftarrow$ *Salarié*.
4. V-t2 $\leftarrow V4$ et **chercher-val**(est-un, E0, V4).
5. réponse $\leftarrow$ *Salarié* et *Etudiant diplômé*.
6. V-t2 $\leftarrow V2$ et **chercher-val**(est-un, E0, V2).
7. réponse $\leftarrow$ *Salarié* et *Etudiant diplômé* et *Salarié*.
8. V-t2 $\leftarrow V1$ et fin.

[4]La complexité de cet algorithme est aussi linéaire en fonction de nombre de couches à parcourir.

Requêtes sur les bases de connaissances

Prenons l'exemple de la requête : *Quels sont les* instances *de la* $4^{ème}$ *version de la classe* Etudiant *de la base* société *du projet* Sherpa*?*.

La première étape de l'exécution de cette requête consiste à identifier la base demandée, à trouver ensuite l'adresse de sa cellule de coopération. Le message contenant la requête est envoyé par la suite à l'adresse trouvée. Pour chaque axe de la base (puisque le nom de l'axe de travail n'a pas été indiqué), le gestionnaire va sélectionner de sa $4^{\text{ème}}$ version la liste des instances de la classe *Etudiant*. L'algorithme est le suivant :

base-demandée ← [**BColl.base-données.(nom-projet="Sherpa") et BColl.base-données.(nom-base="société")**]
adresse-cellule-demandée ← base-demandée.base-données.adresse-base
Pour chaque A ∈ base-demandée.list-axes **faire**
  nom-de-axe ← A.nom-axe
  version-demandée ← A.version.(no-version = 4)
  liste ← version-demandée.couche-correspondante.Etudiant.l-inst
  Écrire (la liste des instances dans la $4^{\text{ème}}$ version de l'axe *nom-de-axe* est = liste)
Fin pour

5.4 Conclusion

Nous avons décrit dans ce chapitre le prototype VOG. Nous avons commencé par le modèle de représentation. Toutes les entités manipulées par notre gestionnaire de versions sont modélisées par des objets. Dès lors, l'entité *Base* est représentée par un objet. Ceci a permis l'utilisateur de manipuler "l'objet" *Base* via son interface fonctionnelle. L'utilisateur peut alors demander de créer l'objet *base*, le modifier et poser des interrogations concernant les attributs de cet objets.

Nous avons ensuite abordé la réalisation des opérations de consultation et de mise à jour des connaissances dans la base. Les conséquences des opérations de mise à jour sont incorporées dans une nouvelle couche. À ce propose, au lieu d'y recopier en entier les entités modifiées, VOG n'y recopie que les informations modifiées de ces entités. Ceci a permis de décharger le contenu des couches.

Dans la dernière section, nous avons exposé le langage de requête. Ce langage contient des opérateurs temporels permettant de poser des requêtes historiques. Le support des versions au niveau des bases nous a facilité la tâche d'exécution des requêtes en général (et des requêtes historiques en particulier). L'explication est donnée par les deux constatations suivantes :

1. Dans la plupart des systèmes de gestion de versions, les versions des objets ont des attributs additionnels par rapport à leur objet et qui servent à les distinguer (voir chapitre 2). Chaque classe de la base, dont la version est gérée, est associée à une autre classe décrivant sa version et faisant également partie de la base. Par exemple, la classe **Personne** est associée à une autre classe **PersonneVersion**. Les attributs de cette classe sont, entre autres, la date de création et le numéro de la version de chaque personne.

   Selon notre modèle de conception, nous n'avons pas eu besoin dans la réalisation de VOG d'utiliser des structures nouvelles afin de représenter les versions des classes et des instances de `Shirka`. Le fait de gérer les versions au niveau des bases nous a mis à l'abri de cette nécessité. Nous avons seulement ajouté une classe supplémentaire **Version** afin de décrire les versions de la classe **Base**. D'où, une classe **Personne** existant dans la $4^{\text{ème}}$ version d'une base n'est que la $4^{\text{ème}}$ version de la classe **Personne**. Dès lors, sa date de création et sa version de dérivation sont les mêmes que celle de la version de la base où elle se trouve.

2. La réponse à certains types de requêtes exige de faire une jointure entre deux (ou plusieurs) classes. Par exemple :
   Quel était le nombre d'étudiants diplômés à Paris en 1991?
   Dans ce cas, il faut joindre les deux versions de classes **Etudiant-diplômé** et **Adresse** créées en 1991 et, regarder pour chaque instance de la première classe si l'instance adresse référencée possède *Paris* comme valeur de l'attribut *ville*. L'interprétation de `VOG` est donc la suivante :
   Pour V $\in$ liste-version(axe-donné) et V.date-création = 1990 **faire**
   nombre-total $\leftarrow$ 0
   Pour chaque instance ED de la classe **Etudiant-diplômé** appartenant à V **faire**
   Pour chaque instance AD de la classe **Adresse** appartenant à V **faire**
   Si ED.adresse = AD et AD.ville = "Paris" **alors**
   nombre-total $\leftarrow$ nombre-total + 1
   Fin Si
   Fin pour
   Fin pour
   Écrire (le nombre total des étudiants diplômés = nombre-total)
   Fin pour

   Joindre deux classes dans une date commune est équivalent à exécuter la jointure dans la couche créée à cette date. Ceci signifie que la jonction temporelle est facile à réaliser dans notre système car les couches représentent déjà des tranches de temps. Pourtant ce type de requêtes considéré comme indispensable par [Wuu et Dayal, 1993] n'est présent que dans quelques modèles (par exemple `TSQL2` [Snodgrass, 1994] et `OODAPLEX` [Wuu et Dayal, 1992]). Une synthèse de ces modèles se trouve dans [Fauvet et Sholl, 1995].

Finalement, la réalisation de notre prototype a été accomplie dans le cadre du modèle `Shirka`. Une des extensions possibles est de l'étendre afin de pouvoir appliquer le versionnement sur l'autre modèle de connaissances (`Tropes`), développé au sein de notre équipe de recherche *Sherpa*.

`Tropes` [Gensel et al., 1994] est un système de représentation de connaissances dont la particularité est d'adjoindre à l'approche classique (comme `Shirka` par exemple) les notions de *concepts* et de *point de vue*. En `Tropes`, une base de connaissances est constituée de *concepts* indépendants. Plusieurs *points de vue* peuvent être associés à un concept, chacun d'eux permettant d'en donner une interprétation particulière. Un point de vue est une hiérarchie de classes dont la racine est une classe représentant toutes les instances du concept. Il existe aussi des relations (appelées *passerelles*) entre classes de points de vue distincts (du même concept). Une passerelle définie entre classe **S** et classe **D** impose l'inclusion des instances de **S** dans **D** (voir fig. 5.10).

Une version de la base de connaissances (codée en `Tropes`) va contenir l'ensemble des concepts et l'ensemble des points de vue regroupés par concept (voir fig. 5.11). Ainsi, l'identificateur d'une entité (concept, classe et instance) n'est plus son nom (cas

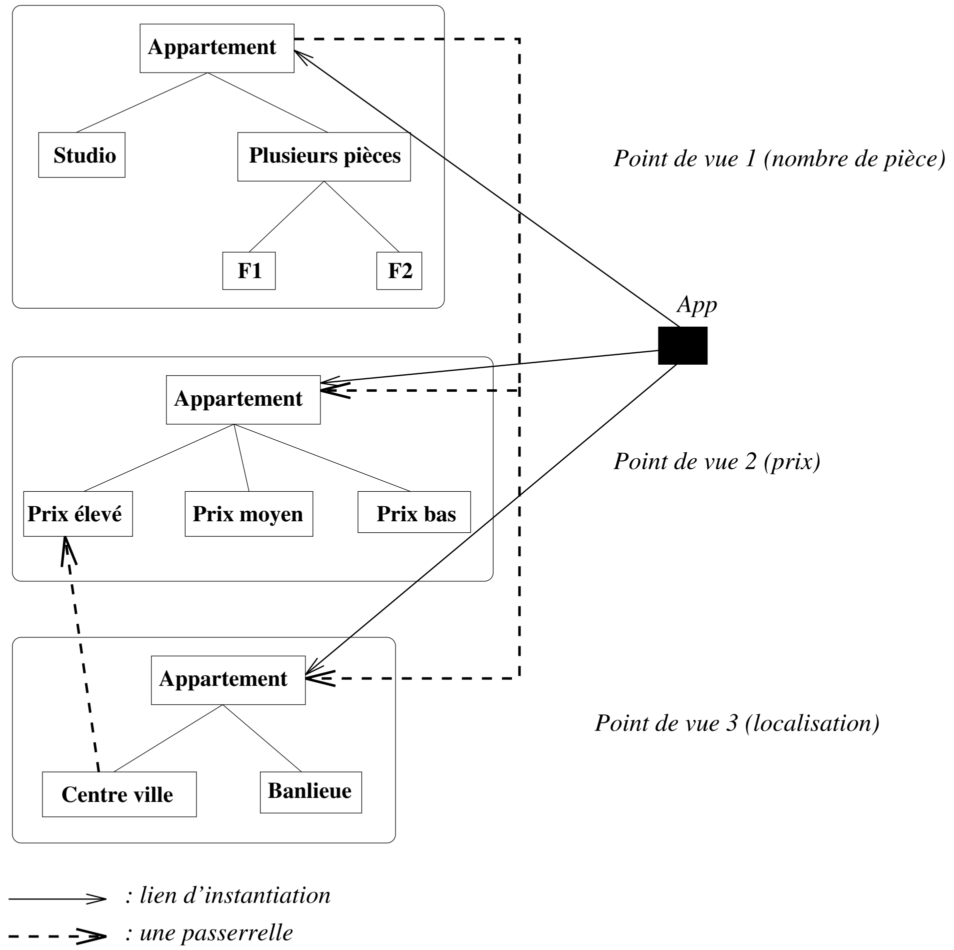


Figure 5.10 - : Le concept **Appartement** est associé en `Tropes` aux trois points de vue : *nombre de pièces*, *localisation* et *prix*. Chaque point de vue organise le concept **Appartement** en une hiérarchie différente. Les trois classes racines sont liées par des passerelles puisqu'elles contiennent toutes les instances du concept **Appartement**. Une passerelle est également définie entre la classe **Centre ville** et **Prix élevé**. Ceci signifie que pour chaque appartement situé au centre ville, son prix est élevé.

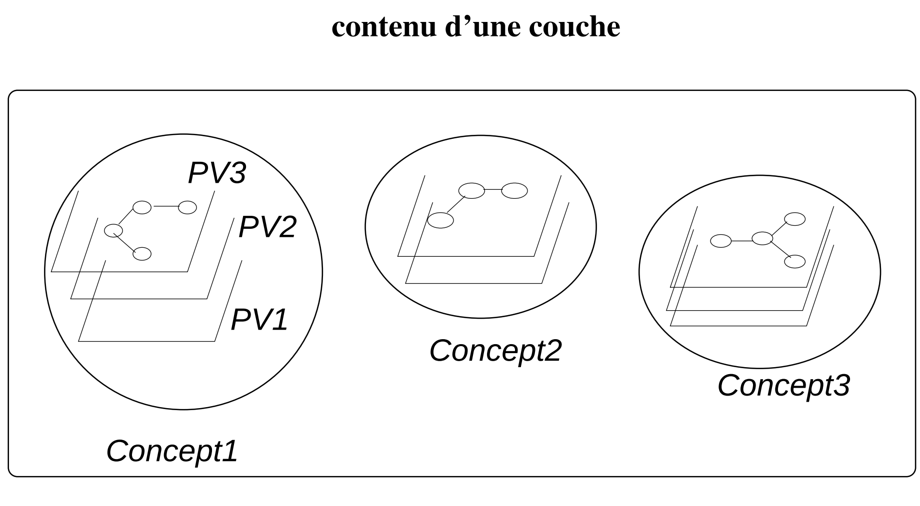


Figure 5.11 - : Une version d'une base de `Tropes`. Elle contient l'ensemble des concepts.

de `Shirka`) mais son nom suivi du nom de son point de vue. Pour changer l'instance *App* (fig. 5.10) l'utilisateur doit déterminer le point de vue selon lequel l'instance concernée est aperçue. Une fois cette détermination faite, on se retrouve alors dans un modèle mono-vue. En ce qui concerne la gestion de la cohérence d'une version (qui dépend fortement du modèle de représentation), cette opération doit prendre en compte les deux contraintes additionnelles : les passerelles et les points de vue. Il faut par exemple étudier les conséquences sur la base d'ajouter ou de supprimer une passerelle, ou bien de définir un nouveau point de vue au sein d'un concept.

Chapitre 6

Conclusion et perspectives

6.1 Problème abordé

Le cadre dans lequel s'inscrit cette thèse est celui de la mise en commun de connaissances détenues par un ensemble de personnes travaillant sur un même domaine (ex: biologie moléculaire). Une telle situation peut typiquement être rencontrée dans le cas de la recherche scientifique lorsque plusieurs équipes de chercheurs travaillent sur un sujet commun et tentent de partager le résultat de leurs expériences. Ces connaissances ne sont pas automatiquement considérées comme correctes et définitives mais peuvent au contraire être mises à jour ou refusées à tout moment par les différentes personnes impliquées. De même, l'ensemble de ces connaissances ne sont pas présentes au moment de la création du système mais elles parviennent au fur et à mesure de leur disponibilité.

La prise en compte des différentes contraintes précédemment énoncées impose une construction incrémentale et partagée d'une base contenant les connaissances consensuelles sur un domaine particulier. Notre objectif a été de proposer une architecture adaptée à ce mode de construction. Cette architecture doit offrir en particulier :

- une gestion de versions multiples de la base de connaissances permettant de garder l'historique des modifications effectuées sur la base.
- une aide à la formulation et au suivi d'hypothèses diverses tout en permettant un retour en arrière aisé en cas d'échec.

6.2 Solution proposée

Afin de répondre au mode de construction de la base auquel nous nous sommes intéressés, nous avons proposé un système de gestion des versions de **bases de connaissances** intégré au sein d'une architecture permettant aux différents chercheurs de réunir leurs connaissances.

6.2.1 Conception et réalisation d'un système de gestion de versions

Une version d'une base dans notre système reflète un *état complet* (l'ensemble des valeurs et des structures de connaissances) de l'évolution de cette base (à un moment donné). Le système que nous avons développé comporte un modèle de représentation de versions et une interface fonctionnelle.

Les versions sont représentées physiquement par des couches. Une couche est créée chaque fois que la base concernée est mise à jour. Elle représente ainsi la différence entre deux versions successives de la base. Ceci permet de comparer facilement ces deux dernières. Cette solution est donc bien adaptée aux bases de connaissances de grandes dimensions puisque le coût mémoire est inférieur à une approche basée sur un stockage extensif de chaque version. De plus, cette solution permet au système de comparer facilement deux versions, qu'elles soient successives ou non. Le résultat de cette opération se trouve dans les couches qui les séparent.

Les différentes couches sont organisées selon un arbre permettant de gérer ainsi les hypothèses multiples. Une branche de cet arbre (appelée axe de travail) représente l'évolution de la base selon l'hypothèse qui est à l'origine de sa création. La longueur de l'arbre reflète l'aspect *incrémental* de la construction et sa largeur (ses différentes branches) reflète l'aspect *partagé et hypothétique* de la construction.

L'interface fonctionnelle développée a pour but de fournir à l'utilisateur un ensemble d'opérations de manipulation des versions de la base de connaissances (création, ajout). Nous nous sommes en particulier intéressés à la définition d'un langage de requêtes. L'utilisateur peut adresser des requêtes concernant les versions de la base (par exemple choisir la version vérifiant des conditions données). Il peut également interroger les valeurs et les structures des connaissances définies dans les différentes versions. Il possède pour cela un ensemble d'opérateurs temporels lui permettant de poser des requêtes historiques (i.e. des requêtes sur les anciennes connaissances). Notre système gérant les versions directement au niveau des bases, la réponse à de telles requêtes est facilitée, une version d'une base étant directement associée à une tranche de temps.

De manière synthétique, le système que nous avons proposé possède les propriétés suivantes :

- **Généralité** : grâce à son indépendance du modèle de représentation de connaissances, il peut être appliqué à n'importe quel modèle de connaissances.

- **Transparence** : grâce à la séparation entre le niveau logique et physique (implémentation), l'utilisateur ne perçoit d'une base que l'ensemble de ses versions. Cette transparence est importante pour permettre aux chercheurs non informaticiens d'utiliser facilement notre système.

Ces deux propriétés permettent en particulier à l'utilisateur, une fois la version déterminée, de se retrouver dans son environnement classique de gestion de base de connaissances mono-version.

6.2.2 Intégration du système de versions dans un environnement d'aide à la construction de la base consensuelle

Nous avons également contribué à la conception de l'environnement de construction de bases. ainsi qu'à l'intégration de notre système de version au sein de cet environnement.

Ce dernier est composé de tous les modules permettant de mettre en œuvre le processus de construction de la base consensuelle :

- gérer les relations entre les différents chercheurs afin de faciliter leur coopération pour obtenir le consensus voulu;
- gérer la base consensuelle ainsi que veiller à sa cohérence;
- examiner l'évolution de la base consensuelle au cours du temps en mettant à la disposition de ces chercheurs les diverses étapes de sa construction.

Ce travail a contribué à réunir les bases de données et les bases de connaissances. Il a mis au service de la base de connaissances des techniques développées essentiellement pour les bases de données. Ces techniques ne sont pas un luxe inutile car, comme nous l'avons montré le long de cette thèse, la dynamicité des bases de connaissances exige la gestion de leurs versions. Il n'existe pas encore à notre connaissance des systèmes qui aient implémenté la gestion de versions dans les bases de connaissances.

6.3 Limites et perspectives

Les perspectives de ce travail sont des extensions possibles de notre système de versions. Ces extensions concernent les points suivants.

Améliorer et étendre l'ensemble d'opérations de manipulation des versions

Notre système offre la possibilité de définir une vue partielle d'une base de connaissances grâce à l'opération de configuration (Cf. §4.4.2). Cette opération est intéressante, mais nécessite toutefois une étude plus approfondie permettant de préciser les problèmes à résoudre pour la réaliser. Il s'agit des problèmes liés à la consistance de la configuration résultante.

Il nous semble également utile d'équiper l'utilisateur d'outils lui permettant de combiner (i.e. fusionner) une version de sa propre base avec une version de la base d'un autre utilisateur ou de la base de son équipe (la BColl dans le cas de notre environnement). Le développement de ces opérateurs doit contrôler, entre autres, la consistance des versions résultantes. Plusieurs anomalies peuvent être détectées (duplication, circulation, contradiction, inclusion, etc.) dans la version d'intégration sans qu'elles soient présentes dans les versions initiales.

Compléter la réalisation de l'environnement de construction de la base consensuelle

Les principaux traits de cette réalisation ont été tracés au cours de cet travail. Cependant, il reste encore beaucoup à faire pour que le potentiel d'un tel environnement puisse être complètement exploité. Mettre en place l'interface graphique semble la première priorité afin de pouvoir gérer les interactions entre les cellules de coopérations et les chercheurs. La spécification et la mise en œuvre des protocoles de communication et d'échange d'informations (genre `KQML`[Finin et al., 1994]) entre les cellules sont également deux tâches essentielles pour l'aboutissement à un prototype fonctionnel.

Annexe A

Exemple de fonctionnement de VOG

Nous allons étudier au cours de cette annexe un exemple de fonctionnement du système de gestion de versions des bases de connaissances. Les traces d'exécution proviennent du dialogue entre l'utilisateur et le système VOG[1].

Comme nous l'avons expliqué au cours de §5.2.1, l'utilisateur doit commencer par la création de sa base (commande *creer-base*) en donnant son nom, et il doit ensuite par l'initialiser (commande *remplir-base*) en chargeant ses propres connaissances ou en réutilisant une version déjà existante. Dans l'exemple suivant, l'utilisateur demande de créer la base *societe* avec l'axe de travail *a1*. Il l'initialise avec ses propres connaissances dont la description est stockée dans le fichier *exemple.sh*. Les classes décrivant ces connaissances sont illustrées figure A.1.

```
module-version : creer-base
nom-base ? societe
nom-axe-travail ? a1
module-version : remplir-base
fichier-ou-version ? fichier
nom-fichier ? exemple.sh
Chargement du fichier
/homes/daulagiri/sherpa/tayar/Shirka/BasesShirka/exemple.sh
  dans la base societe
-> /homes/daulagiri/sherpa/tayar/Shirka/BasesShirka/exemple.sh
```

Dans le deuxième exemple, l'utilisateur charge la base *societe* dans son espace de travail (commande *charger-base*). Il demande ensuite de la manipuler (commande *ouvrir-base*). Cette commande signale le début d'une session de consultation de la base concernée. Si l'utilisateur souhaite modifier celle-ci, il doit taper la commande *mise-a-jour* et en précisant le nom de l'axe et le numéro de la version à modifier. Dans cet exemple suivant, l'utilisateur choisit de mettre à jour la version courante (le symbole * désigne la version courante) de l'axe *a1*. Nous considérons désormais que l'axe *a1* est l'axe de travail courant.

```
module-version : charger-base
base ? societe
module-version : ouvrir-base
base ? societe
```

[1]Le premier mot suivit du caractère ":" ou "?" correspond à un message système. Le reste de la ligne correspond à la donnée fournie par l'utilisateur.

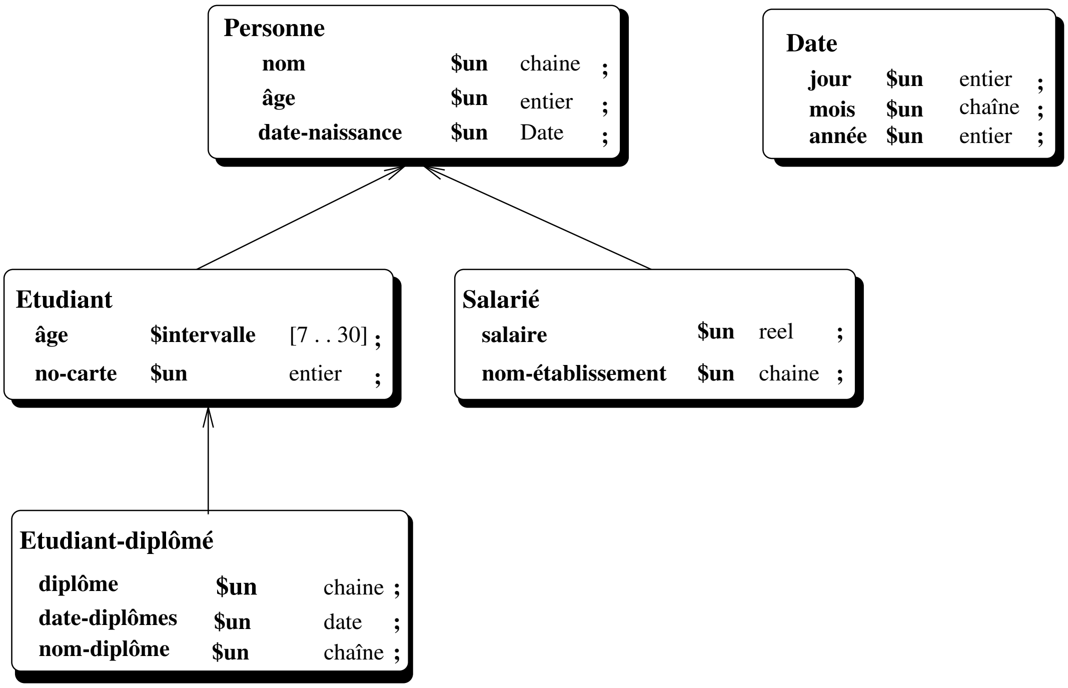


Figure A.1 - : Les classes qui décrivent les connaissances de la base *société*.

```
module-version : mise-a-jour
nom-axe-travail ? a1
version ? *
```

À l'issue de la commande *mise-à-jour*, une session de travail est déclenchée et une nouvelle version est créée. En tapant la commande *liste-version*, le système rend la liste des versions ordonnées par leur date de création. Le contexte de travail courant est l'axe *a1* et sa version courante *version2*. L'utilisateur peut effectuer dans celle-ci les modifications voulues. Dans l'exemple suivant, il incorpore l'instance *p0* de la classe **Personne** en fournissant des valeurs pour les deux attributs *nom* et *âge*,ainsi que l'instance *s0* de la classe **salarié** qui elle ne contient pas de valeurs pour ses attributs. La session est terminée par la commande *sauver-base*. La 2$^{\text{ème}}$ version de la base est alors figée.

```
module-version : mise-a-jour
module-version : liste-version
  La liste de versions de la base societe (ordonnees
  par leur date de creation) est la suivante :
 version1  est creee le  (Sat July  1 95 12:28:44 )
 version2  est creee le  (Sat July  1 95 12:33:09 )
module-version : cr-inst
nom-classe ? personne
Classe : personne
Sous-classe ? -
Nom de l'instance ? p0
nom ? "ntayar"
age ? 28
date-naissance ? <
-> p0
```

```
module-version : cr-inst
nom-classe ? salarie
Classe : salarie
Nom de l'instance ? s0
nom ? <
-> s0
```

Dans l'exemple suivant, l'utilisateur modifie de nouveau la base. Une 3$^{\text{ème}}$ version est créée à l'issue de la commande *mise à jour*. Dans cette version, l'utilisateur effectue les manipulations suivantes :

1. création de l'instance *d0* de la classe **Date**. Cette instance possède les valeurs : *jour=15*, *mois ="septembre"* et *année=1966*;

2. addition de la valeur de l'attribut *date-naissance* dans la classe **Personne**. Cette valeur est une référence à l'instance créée *d0*;

3. création de l'instance *e0* appartenant à la classe **Etudiant**.

```
module-version : mise-a-jour
module-version : liste-version
  La liste de versions de la base societe (ordonnees
  par leur date de creation) est la suivante :
 version1  est creee le  (Sat July  1 95 12:28:44 )
 version2  est creee le  (Sat July  1 95 12:33:44 )
 version3  est creee le  (Sat July  1 95 12:40:54 )
module-version : cr-inst
nom-classe ? date
Classe : date
Nom de l'instance ? d0
jour ? 15
mois ? "septembre"
annee ? 1966
-> d0
module-version : aj-val
instance ? p0
attribut ? date-naissance
valeur ? d0
-> d0
module-version : cr-inst
nom-classe ? etudiant
Classe : etudiant
Sous-classe ? -
Nom de l'instance ? e0
nom ? <
-> e0
module-version : sauver-base
```

L'exemple suivant correspond à une consultation de l'historique des changements effectués. L'utilisateur demande de visualiser l'instance *p0* (commande *vi*). Dans la version courante, cette instance a les valeurs : *nom*="ntayar", *âge*=28 et *date-naissance*=d0. Dans la version précédente, seules les deux premières valeurs sont présentes, la date de naissance ayant été fournie dans la version courante (voir exemple précédent).

```
module-version : vi
instance ? p0
niveaux ? -
 Donnez le numero de la version que vous desirez?
 S'il s'agit de la version courante, taper *
no-version? *
{ p0
     est-un = personne ;
     nom = "ntayar";
     age = 28 ;
     date-naissance = d0 ;
}
-> t
module-version : vi-inst
instance ? p0
niveaux ? -
 Donnez le numero de la version que vous desirez?
 S'il s'agit de la version courante, taper *
no-version? 2
{ p0
     est-un   =  personne;
     nom = "ntayar";
     age = 28 ;
}
-> p0
```

La suivante interrogation concerne la liste des instances de la classe **Personne** (commande *l-inst*). Cette liste dépend de la version où la classe concernée se trouve. Dans la version courante, la classe **Personne** a gagné l'instance *e0* de la classe **Etudiant** (selon le principe d'inclusion des instances). La commande *l-inst* appliquée à la classe **Personne** dans la version courante rend donc la liste des instances *p0, s0* et *e0*. Cette même commande appliquée à la même classe dans la version précédente (i.e. version no. 2) ne rend que la liste des instances *p0* et *s0*.

```
module-version : l-inst
classe ? personne
version ?  *
-> p0 e0 s0
module-version : l-inst
classe ? personne
version ? 2
-> p0 s0
```

L'exemple suivant montre une nouvelle session de modification (le nombre de version créée est donc maintenant égal à 4). L'utilisateur demande de : (1) supprimer la valeur de l'attribut *âge* de l'instance *p0* (commande *sup-val*) et (2) de modifier la valeur de l'attribut *jour* de l'instance *d0* (commande *mod-val*).

```
module-version : mise-a-jour
module-version : liste-version
  La liste de versions de la base societe (ordonnees
  par leur date de creation) est la suivante :
 version1  est creee le  (Sat July  1 95 12:28:44 )
 version2  est creee le  (Sat July  1 95 12:33:44 )
 version3  est creee le  (Sat July  1 95 12:40:44 )
 version4  est creee le  (Sat July  1 95 12:45:02 )
module-version : sup-val
instance ? p0
attribut ? age
valeur ? -
-> 28
module-version : mod-val
instance ? d0
attribut ? jour
valeur ? 21
ancienne-valeur ? -
-> 21
module-version : sauver-base
```

L'exemple suivant permet de vérifier grâce à la visualisation que l'attribut *âge* de *p0* n'existe plus dans la version courante mais est néanmoins toujours présent dans la version précédente (référencée par -1).

```
module-version : vi
instance ? p0
niveaux ? -
 Donnez le numero de la version que vous desirez?
 S'il s'agit de la version courante, taper -
no-version? -
{ p0
     est-un = personne ;
     nom = "ntayar";
     date-naissance =  d0 ;
}
-> t
module-version : vi
instance ? p0
niveaux ? -
 Donnez le numero de la version que vous desirez?
 S'il s'agit de la version courante, taper -
no-version? -1
{ p0
     est-un   =  personne;
     nom = "ntayar";
```

```
     age = 28 ;
     date-naissance =  d0 ;
}
-> p0
```

L'utilisateur pose ensuite une requête de consultation concernant les valeurs de l'attribut *jour* de l'instance *d0*. Les réponses fournies par le système (et données dans l'exemple ci-dessous) sont :

- pour la version courante (version no.4), la valeur de cet attribut est 21;
- pour la $3^{\text{ème}}$ version, la valeur est 15;
- pour la version no. 2, un message d'erreur est affiché signalant que l'instance concernée *d0* n'existe pas dans cette version (puisqu'elle a été créée dans la $3^{\text{ème}}$ version, au cours du quatrième exemple de cette annexe).

```
module-version : val?
instance ? d0
attribut ? jour
 Donnez le numero de la version que vous desirez?
 S'il s'agit de la version courante, taper *
no-version?  *
-> 21
module-version : val?
instance ? d0
attribut ? jour
 Donnez le numero de la version que vous desirez?
 S'il s'agit de la version courante, taper *
no-version?  3
-> 15
module-version : val?
instance ? d0
attribut ? jour
 Donnez le numero de la version que vous desirez?
 S'il s'agit de la version courante, taper *
no-version?  2
erreur, l'entite d0  n'existe pas dans la version 2
 de l'axe de travail a1 de la base societe!
```

L'utilisateur commence une cinquième session de modification (la $5^{\text{ème}}$ version est ainsi créée). Il demande de classer l'instance *e0* dans la classe **Etudiant-diplômé**. Pour cela, le système lui demande de donner des valeurs aux attributs impliqués dans la classification (i.e. les attributs de la classe **Etudiant-diplômé**). Selon les valeurs fournies, le système détermine si **Etudiant-diplômé** est une classe sûre, possible ou impossibles. Dans le cas de l'exemple suivant, la classe **Etudiant-diplômé** est jugée comme sûre (i.e. elle peut recevoir l'instance *e0*). L'utilisateur demande alors de rattacher *e0* à cette classe. Il sauve enfin cette session.

```
module-version : mise-a-jour
```

```
module-version : liste-version
La liste de versions de la base societe (ordonnees
  par leur date de creation) est la suivante :
 version1  est creee le  (Sat July  1 95 12:28:44 )
 version2  est creee le  (Sat July  1 95 12:33:44 )
 version3  est creee le  (Sat July  1 95 12:40:44 )
 version4  est creee le  (Sat July  1 95 12:45:02 )
 version5  est creee le  (Sat July  1 95 13:00:02 )
classif : classer-inst
instance ? e0
classe ? etudiant-diplome
classification de l'instance e0 a partir de la classe
etudiant-diplome
no-diplome ? 4
date-diplome ? d0
nom-diplome ? "these"
les classes sures:
-> etudiant-diplome
classif : ratt-inst
instance ? e0
classe ? etudiant-diplome
-> etudiant-diplome
module-version : sauver-base
```

À l'issue de ces opérations de modifications, la classe **Etudiant-diplômé** a eu une instance de plus (*e0*) dans la version courante. Cette classe ne possède aucune instance pour la version précédente.

```
module-version : l-inst
classe ? etudiant-diplome
version ? *
->  e0
module-version : l-inst
classe ? etudiant-diplome
version ? precedente
-> echec
```

Le dernier exemple concerne la manipulation des classes. L'utilisateur demande, au cours d'une nouvelle session, d'ajouter la classe **Chômeur** à la hiérarchie initiale des classes. Cette classe est déclarée comme sous-classe de **Personne**, et possède deux attributs *début-chômage* et *carte-chômeur*. L'utilisateur la définit en utilisant la fonction `Le-Lisp` *def-sh*. Afin d'insérer cette classe dans la version courante, il utilise la commande *aj-classe*.

```
(def-sh '(chômeur
        (sorte-de       (= personne))
        (début-chômage  (\$un date))
        (carte-chômeur  (\$un entier))))

module-version : aj-classe
```

```
classe ? chômeur
-> classe chômeur est ajoutée à la base societe
```

Après cette insertion, la liste des sous-classes (commande *l-spec*) de la classe **Personne** possède la classe **chômeur** dans la version courante. Cette nouvelle classe ne faisait pas partie de cette liste dans les versions précédentes (car elle n'existait pas).

```
module-version : l-spec
classe ? personne
version ? *
niveaux ? -
etudiant
    etudiant-diplome
salarie
chômeur
-> personne
module-version : l-spec
classe ? personne
version ? -1
niveaux ? -
etudiant
    etudiant-diplome
salarie
-> personne
```

Table des figures

Bibliographie

[Adiba et al., 1987] Adiba, M., Quang, N. B., et Collet, C. (1987). Aspects temporels, historiques et dynamiques des bases de données. *Techniques et sciences informatiques (TSI)*, 6(5):457–478.

[Agrawal et Sengupta, 1989] Agrawal, D. et Sengupta, S. (1989). Modular synchonization in multiversion databases : Version control and concurrency control. Dans *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 408–417.

[Agrawal et al., 1991] Agrawal, R., Buroff, S. J., Gehani, N. H., et Shasha, D. (1991). Object Versioning in Ode. Dans *Proceedings of the 7th IEEE Conference on Data Engineering*.

[Ahmed et Navathe, 1991] Ahmed, R. et Navathe, S. B. (1991). Version management of composite objects in CAD databases. Dans Clifford, J. et King, R., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 218–227, Colorado (Etats-Unis).

[Ahmed-Nacer, 1994] Ahmed-Nacer, M. (1994). *Un modèle de gestion et d'évolution de schéma pour les bases de données de génie logiciel*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble - INPG, Grenoble (France).

[Al-Jadir et al., 1993] Al-Jadir, L., Falquet, G., et Léonard, M. (1993). Context Versions in an Object-Oriented Model. Dans Marik, V., Lazansky, J., et Wagner, R. R., editors, *Proceedings of the 4th International Conference on Database and Expert Systems Applications (DEXA)*, pages 24–35, Prague (Republique Tchèque). Springer-Verlag. Lecture notes in computer science.

[Andany et al., 1991] Andany, J., Léonard, M., et Palisser, C. (1991). Management of schema evolution in databases. Dans *Proceedings of the International Conference on VLDB*, pages 161–170, Barcelone (Espagne).

[Barbian et Schlageter, 1993] Barbian, G. et Schlageter, G. (1993). Coda -a groupbase-system for cooperative design applications. Dans *Proceedings of the 1st International Conference on Intelligent and Cooperative Information Systems (ICICIS)*. IEEE Computer Society Press.

[Beech et Mahbod, 1988] Beech, D. et Mahbod, B. (1988). Generalized version control in an object-oriented database. Dans *Proceedings of IEEE Conference on Data Engineering*, pages 14–22, Los Angeles (Etats-Unis).

[Belkhatir, 1988] Belkhatir, N. (1988). *Nomade: un noyau d'environnement pour la programmation globale*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble - INPG, Grenoble (France).

[Belkhatir et al., 1991] Belkhatir, N., Estublier, J., et Melo, W. L. (1991). Adele2 : a support to large software development process. Dans *Proceedings of the 1st Conference on Software Process (ICSPI)*, pages 159–170, Redondo Beach (Etats-Unis).

[Bernstein et al., 1987] Bernstein, P. A., Hadzilacos, V., et Goodman, N. (1987). *Concurrency control and recovery in database systems.* Addison-Wesley.

[Björnerstedt et Britts, 1988] Björnerstedt, A. et Britts, S. (1988). AVANCE : an object management system. Dans Meyrowitz, N., editor, *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 206–221, San Diego (Etats-Unis).

[Björnerstedt et Hultén, 1989] Björnerstedt, A. et Hultén, C. (1989). Version control in an object-oriented architecture. Dans Kim, W. et Lochovsky, F. H., editors, *Object oriented concepts, database and applications*, pages 451–485. ACM Press, New York (Etats-Unis).

[Boissier, 1990] Boissier, O. (1990). La coopération entre systèmes à base de connaissances. Technical Report RR-811-I, IMAG - LIFIA.

[Brachman et al., 1983] Brachman, R. J., Fikes, R. E., et Levesque, H. J. (1983). Krypton: A functional approach to knowledge representation. *Computer*, 16(10):67–73.

[Bubenko, 1977] Bubenko, J. A. (1977). The temporal dimension in information modeling. Dans *Architecture and models in database management systems.*

[Capponi, 1992] Capponi, C. (1992). Acquisition de connaissances et dynamique des classes dans une représentation de connaissances par objets. DEA informatique, Université Joseph Fourier, Grenoble (France).

[Capponi, 1995] Capponi, C. (1995). *Identification et exploitation des types dans un modèle de représentation de connaissances par objets.* Thèse de doctorat en informatique, Université de Joseph Fourier, Grenoble (France).

[Casais, 1990] Casais, E. (1990). *Managing evolution in object oriented environments : an algorithmic approach.* Thèse de doctorat, Université de Genève, Genève (Suisse).

[Cellary et Jomier, 1990] Cellary, W. et Jomier, G. (1990). Consistency of versions in object-oriented databases. Dans McLeod, D., Sacks-Davis, R., et Schek, H., editors, *Proceedings of The International Conference on VLDB*, pages 432–441, Brisbane (Australie).

[Cellary et Jomier, 1994] Cellary, W. et Jomier, G. (1994). Apparent Versioning and Concurrency Control in Object-Oriented Databases. Dans *Proceedings of the International Conference on Computing and Information*, Peterborough (Canada).

[Chevalier, 1994] Chevalier, C. (1994). Construction incrémentale de bases de connaissances à objets. Mémoire d'ingénieur CNAM, CUEFA.

[Chou et Kim, 1986] Chou, H. T. et Kim, W. (1986). A unifying framework for version control in CAD. Dans *Proceedings of the 12th conference on VLDB*, pages 336–344, Kyoto (Japon).

[Chou et Kim, 1988] Chou, H. T. et Kim, W. (1988). Versions and change notification in an object-oriented database system. Dans *Proceedings of the ACM/IEEE Design Automation Conference*.

[Crampé, 1995] Crampé, I. (1995). Suggestion de révisions dans une base de connaissances à objets. Rapport interne, INRIA Rhône-Alpes, Grenoble (France).

[Dittrich, 1989] Dittrich, K. (1989). The Damokles database system for design applications ; its past, its present, and its future. Dans Bennett, K. H., editor, *Software Engineering Environments : Research and Practice*, pages 151–171. E. horwood Books, Durhan (Royaume-Uni).

[Dittrich et Lorie, 1988] Dittrich, K. et Lorie, R. (1988). Version support for engineering database systems. *IEEE transactions on software engineering*, 14(4):429–437.

[Dittrich et al., 1986] Dittrich, K. R., Gotthard, W., et Lockemann, P. C. (1986). DAMOKLES - a database system for software engineering environments. Dans Goos, G. et Hartmanis, J., editors, *Proceedings of an International Workshop on Advanced programming Environments*, pages 354–371, Trondheim (Norvège). Springer-Verlag. Lecture notes in computer science.

[Ellis et al., 1991] Ellis, C., Gibbs, S. J., et Rein, G. L. (1991). Groupware, some issues and experiences. *Communications of the ACM*, 34(1):39–58.

[Estublier, 1985] Estublier, J. (1985). A configuration manager : the Adele data base of programs. Dans *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 140–147, Massachusetts (Etats-Unis).

[Estublier et Casallas, 1994] Estublier, J. et Casallas, R. (1994). The Adele configuration manager. Dans Tichy, W., editor, *Configuration Management*, Software Trend Serie. Wiley and son.

[Euzenat, 1995] Euzenat, J. (1995). Building consensual knowledge bases: context and architecture. Dans Mars, N., editor, *Towards Very Large Knowledge bases (proc. of the 2nd International Conference on Building and Sharing very Large-scale Knowledge Bases)*, pages 143–155, Enschede (Pays - Bas). IOS Press.

[Falquet, 1990] Falquet, G. (1990). F2 an object-oriented database model with semantic contextes. CUI Research Report - Genève.

[Fauvet, 1988] Fauvet, M. C. (1988). *ETIC : un SGBD pour la CAO dans un environnement partagé*. Thèse de doctorat en informatique, Université de Jospeh Fourier, Grenoble (France).

[Fauvet, 1989] Fauvet, M. C. (1989). Définition et réalisation d'un modèle de versions d'objets. Dans INRIA, editor, *Actes des 5ème journées bases de données avancées*, pages 269–289, Genève (Suisse).

[Fauvet et Sholl, 1995] Fauvet, M. C. et Sholl, P. C. (1995). Temps et bases de données : concepts temporels pour la gestion de l'évolution des données. Rapport de recherche RR 945 I, LGI - IMAG, Grenoble (France).

[Finin et al., 1994] Finin, T., Fritzson, R., MacKay, D., et MacEntire, R. (1994). KQML as an agent communication language. Technical report CS-94-02, Enterprise Integration Technologies (EIT), University of Maryland (Etats-Unis).

[Gadia, 1993] Gadia, S. (1993). Ben-zvi's poineering work in relational temporal databases. Dans *Temporal Databases*. The Benjamins/Cummings Publishing Company.

[Gadia et Yeung, 1988] Gadia, S. K. et Yeung, C. S. (1988). Inadequacy of interval time stamps in temporal databases. Dans *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 251–259, Chicago (Etats-Unis).

[Gambetta, 1993] Gambetta, W. (1993). Combining multiple knowledge bases. Dans *Proceedings of the IJCAI Worshop on Knowledge Sharing and Information Interchange*, pages 22–25, Chambery (France).

[Gançarski, 1994] Gançarski, S. (1994). *Versions et bases de données : modèle formel, supports de langage et d'interface-utilisateur.* Thèse de doctorat en science, Université Paris-Sud, Paris (France).

[Gensel et al., 1994] Gensel, J., Girard, P., et Schmeltzer, O. (1994). Intégration de contraintes, d'objets composites et de tâches dans un modèle de représentation de connaissances par objets. Dans *Actes de la Conférence Reconnaissance des Formes et Intelligence Artificielle (RFIA)*, pages 281–292, Paris (France).

[Goldstein et Bobrow, 1980] Goldstein, I. P. et Bobrow, D. G. (1980). Description for a programming environment. Dans *Proceeding of AAAI*, pages 187–189, Stanford (Etats-Unis).

[Grivaud, 1992] Grivaud, S. (1992). Navigation dans une base de connaissances à objets. Mémoire d'ingénieur CNAM, CUEFA, Grenoble(France).

[Haton et al., 1991] Haton, J. P., Bouzid, N., Charpillet, F., Haton, M. C., Lâasri, B., Lâasri, H., Marquis, P., Mondot, T., et Napoli, A. (1991). *Le raisonnement en intelligence artificielle.* Collection iia (informatique et intelligence artificielle). InterEditions, Paris (France).

[Ilog, 1991] Ilog (1991). *Le-Lisp de l'INRIA.* Société ILOG, Gentilly (France). Manuel de référence, Version 15.24.

[Jarke et al., 1993] Jarke, M., Jeusfeld, M. A., et Szczurko, P. (1993). Three aspects of intelligent cooperation in the quality life cycle. *International Journal of Intelligent and Cooperative Information Systems.*

[Kafer et Achöning, 1992] Kafer, W. et Achöning, H. (1992). Mapping a version model to a complex-object data model. Dans *Proceedings of International Conference on Data Engineering*, Tempe (Etats-Unis).

[Kaiser et Habermann, 1983] Kaiser, G. E. et Habermann, A. N. (1983). An environment for system version control. Dans *Digest of papers COMPCOM.* IEEE Computer Society, San Franscico (Etats-Unis).

[Katz et Lehmann, 1984] Katz, R. et Lehmann, T. (1984). Database support for versions and alternatives of large design files. *IEEE Transactions on Software Engineering Conference*, SE 10(2).

[Katz, 1990] Katz, R. H. (1990). Toward a unified framework for version modeling. *ACM Computing Surveys*.

[Kim et al., 1989] Kim, W., Ballou, N., Chou, H. T., Garza, J. F., et Woelk, D. (1989). Features of the Orion object-oriented database system. Dans Kim, W. et Lochovsky, F. M., editors, *Object-oriented concepts, databases and applications*, Frontier series. ACM Press.

[Kim et al., 1987] Kim, W., Banerjee, H., Garza, J., et Woelk, D. (1987). Composite object support in object-oriented database system. Dans *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA)*, Orlando (Etats-Unis).

[Kim et al., 1990] Kim, W., Banerjee, J., Chou, H., et Garza, J. F. (1990). Object-oriented database support for CAD. *Butterworth-Heinemann, Computer-aided design*, 22(8):469–479.

[Kim et Batory, 1985] Kim, W. et Batory, D. (1985). A model and storage technique for versions of VLSI CAD objects. Dans *Proceedings of the Conference on Foundations of Data Organization*, Kyoto (Japon).

[Kim et Chou, 1988] Kim, W. et Chou, H. T. (1988). Versions of schema for object-oriented databases. Dans *Proceedings of the International Conference on VLDB*, pages 148–159, Los Angeles (Etats-Unis).

[Klopprogge et Lockemann, 1983] Klopprogge, M. R. et Lockemann, P. C. (1983). Modelling information preserving databases: consequences of the concept of time. Dans *Proceedings of the Conference on VLDB*, pages 399–416.

[Lemaire, 1993] Lemaire, F. (1993). Etude des protocoles d'intéraction dans un environnement d'aide à la recherche collaborative. DEA sciences Cognitives, INPG, Grenoble (France).

[Lemaire et Tayar, 1994] Lemaire, F. et Tayar, N. (1994). Gestion de la construction incrémentale d'une base de connaissances consensuelles. Dans *Actes 2nd rencontres nationales des jeunes chercheurs en intelligence artificielle*, Marseille (France).

[Mahé, 1994] Mahé, S. (1994). Des collecticiels à la construction concourante de bases de connaissances. DEA système d'information, INPG, Grenoble (France).

[Mariño, 1993] Mariño, O. (1993). *Classification dans les systèmes de représentation à objets*. Thèse de doctorat en informatique, Université Joseph Fourier, Grenoble (France).

[Masini et al., 1989] Masini, G., Napoli, A., Colnet, D., Léonard, D., et Tombre, K. (1989). *Les langages à objets*. InterEditions.

[Mattos, 1988] Mattos, N. M. (1988). KRISYS - A Multi-Layered Prototype KBMS Supporting Knowledeg Independence. Dans *Proceedings of the International Computer Science Conference - Artificial Intelligence : Theory and Application*, pages 31–38, Hong-Kong.

[Melo, 1993] Melo, W. M. (1993). *TEMPO : un environnement de développement logiciel centré procédés de fabrication*. Thèse de doctorat en informatique, Université Joseph Fourier, Grenoble (France).

[Munch, 1993] Munch, B. P. (1993). *Versioning in a software engineering database - the change oriented way*. Thèse de doctorat en informatique, Université de Trondheim, Norvège.

[Mylopoulos et al., 1990] Mylopoulos, J., Borgida, A., Jarke, M., et Koubarakis, M. (1990). Telos: a language for representation knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362.

[Nguyen et Rieu, 1989] Nguyen, G. T. et Rieu, D. (1989). Schema change propagation in object-oriented databases. Dans Ritter, G. X., editor, *Proceedings of International Federation on Information Processing*, pages 815–820. Elsevier Science Publisher.

[Odberg, 1992] Odberg, E. (1992). A framework for managing schema versioning object-oriented databases. Dans Tjoa, A. M. et Ramos, I., editors, *Proceedings of Database and Expert Systems Applications (DEXA)*, pages 115–120. Springer-Verlag.

[Palisser, 1990] Palisser, C. (1990). Le modèle de versions du système Charly. Dans *Actes des 6èmes journées Bases de Données avancées*, Montpellier (France). INRIA.

[Penny et Stein, 1987] Penny, J. D. et Stein, J. (1987). Class modification in the GemStone object-oriented DBMS. Dans *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 111–117, Orlando (Etats-Unis).

[Plaice et Wadge, 1993] Plaice, J. et Wadge, W. (1993). A new approach to version control. *IEEE Transaction on Software Engineering*, 19(3):268–275.

[Quang, 1986] Quang, N. B. (1986). *Aspects dynamiques et gestion du temps dans les systemes de bases de données généralisées*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble - INPG, Grenoble (France).

[Rechenmann, 1988] Rechenmann, F. (1988). Shirka : un modèle de connaissances centré-objet. Dans *Organisation et traitemant des connaissances en intelligence artificielle*, actes d'université d'été 7. CIRILLE, Lyon (France).

[Rechenmann, 1993] Rechenmann, F. (1993). Building and sharing large knowledge bases in molecular genetics. Dans *Proceedings of the 1st International Conference on Building and Sharing Very Large-Scale Knowledge Bases*, pages 291–301, Tokyo (Japon).

[Rechenmann, 1994] Rechenmann, F. (1994). Object-oriented modeling of biological knowledge. Dans *Actes des 7ème Entretiens du Centre Jacques Cartier*, Informatique et biologie moléculaire, pages 105–109, Lyon (France).

[Rechenmann et al., 1990] Rechenmann, F., Fontanille, P., et Uvietta, P. (1990). *SHIRKA : système de gestion de bases de connaissances centrées-objet.* INRIA Rhône-Alpes, Grenoble (France). Manuel d'utilisation.

[Reichenberger, 1989] Reichenberger, C. (1989). Orthogonal version management. Dans *Proceedings of the Third International Workshop on Software Configuration Management*, pages 137–140, New Jersey (Etats-Unis).

[Rochkind, 1975] Rochkind, M. J. (1975). The source code control system. *IEEE Transaction on Software Engineering*, SE-1(4):364–370.

[Schmeltzer et al., 1993] Schmeltzer, O., Médigue, C., Uvietta, P., Rechenmann, F., Dorkeld, F., Perrière, G., et Gautier, C. (1993). Building large knowledge bases in molecular biology. Dans Hunter, L., Searls, D., et Shavlik, J., editors, *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology*, pages 345–353, Washington D.C - USA. AAAI Press.

[Sciore, 1991] Sciore, E. (1991). Multidimensional Versioning for Object-Oriented Databases. Dans *Proceedings of the Second International Conference on Deductive and Object-Oriented Database.*

[Sciore, 1994] Sciore, E. (1994). Versioning and configuration management in an object-oriented data model. *VLDB journal*, 3(1):77–106.

[Skarra et Zdonik, 1987] Skarra, A. H. et Zdonik, S. B. (1987). Type evolution in an object-oriented database. Dans Shriver, B. et Wegner, P., editors, *Research directions in object-oriented programming*, pages 393–415. MIT Press.

[Snodgrass, 1994] Snodgrass, R. (1994). TSQL2 language specification. *ACM SIGMOD Record*, 23(11).

[Streitz et al., 1992] Streitz, N., Haake, J., Hannemann, J., Lemke, A., Schuler, W., Schutt, H., et M.Thuring (1992). Sepia : a cooperative hypermedia authoring environment. Dans *Proc. of the ACM Conference on HyperText*, pages 11–21. ACM Press.

[Talens, 1994] Talens, G. (1994). *Gestion de versions d'objets simples et composites.* Thèse de doctorat en génie informatique, Université de Montpellier II - Sciences et techniques du languedoc, Montpellier (France).

[Talens et al., 1993] Talens, G., Oussalah, C., et Colinas, M. E. (1993). Versions of simple and composite objects. Dans *Proceedings of the International Conference on VLDB*, pages 62–72, Dublin (Irlande).

[Tansel et al., 1993] Tansel, A., Clifford, J., Gadia, S., Jajodia, S., Segev, A., et Snodgrass, R., editors (1993). *Temporal Databases : Theory, Design and Implementation.* Data Systems and Applications. The Benjamin / Cummings Publishing Company, Redwood city (Etats-Unis).

[Tayar, 1993] Tayar, N. (1993). A model for developing large shared knowledge bases. Dans Bhargava, B., Finin, T., et Yesha, Y., editors, *Proceedings of the Second International Conference on Information and Knowledge Management*, Washington D. C. (Etats-Unis). ACM Press.

[Tichy, 1982] Tichy, W. (1982). Design, implementation and evaluation of a revision control system. Dans *Proceedings of the 6th Conference on Software Engineering*, pages 58–67, Tokyo (Japon). IEEE Computer Society.

[Tichy, 1988] Tichy, W. (1988). Tools for software configuration management. Dans *Proceedings of the1st International Workshop on Software Version and Configuration Control*, Stuttgart (Allemagne).

[Trousse, 1989] Trousse, B. (1989). *Coopération entre systèmes à base de connaissances et outils de CAO : l'environnement multi-agent ANAXAGORE*. Thèse de doctorat en sciences, Université de Nice Sophia-Antipolis, Nice (France).

[Tuzhilin et Clifford, 1990] Tuzhilin, A. et Clifford, J. (1990). A temporal relation algebra as a basis for temporal relational completness. Dans *Proceedings of the Conference on VLDB*, pages 13–23, Brisbane (Australia).

[Weihl, 1987] Weihl, W. E. (1987). Distributed version management for read-only actions. *IEEE Transactions on Software Engineering*, SE-13(1):55–64.

[Werner et Demazeau, 1992] Werner, E. et Demazeau, Y., editors (1992). *Decentralized A. I. 3*. Elsevier Science Publishers B. V., Amsterdam (Pays-Bas).

[Wiebe, 1993] Wiebe, D. (1993). Object-Oriented Software Configuration Management. Dans *Proceedings of 4th Software Configuration Management Workshop*, Baltimore (Etats-Unis).

[Wiederhold et al., 1975] Wiederhold, G., Fries, J., et Weyl, S. (1975). Structured organization of clinical databases. Dans *Proceedings of AFIPS National Computer Conference*.

[Wuu et Dayal, 1992] Wuu, G. T. J. et Dayal, U. (1992). A uniformal model for temporal object-oriented databases. Dans *Proceedings of IEEE Conference on Data Engineering*.

[Wuu et Dayal, 1993] Wuu, G. T. J. et Dayal, U. (1993). A uniform Model for Temporal and Versioned Object-Oriented Databases. Dans Tansel, U. A., Clifford, J., Gadia, S., Jajodia, S., Segev, A., et Snodgrass, R., editors, *Temporal Databases : Theory, Design and Implementation*, Database Systems and Applications, chapter 10, pages 230–247. Benjamin/Cummings, Redwood City (Etats-Unis).

[Zdonik, 1986] Zdonik, S. B. (1986). Version Management in an Object-Oriented Database. Dans Conradi, R., Didriksen, T., et d. Wanvik, editors, *Lecture notes in computer science, International Workshop on Advanced Programming Environnements*, pages 405–422, Throndheim (Norvège).

[Zicari, 1991] Zicari, R. (1991). A framework for schema updates in an object oriented database system. Dans *Proceedings of the International Conference on Data Engineering*, pages 2–13, Kobe (Japon).